

**Proseminar Websprachen  
WS 2003/2004**

Perl

Andreas Bierfert

7. Dezember 2003

**FINAL DRAFT**

## Inhaltsverzeichnis

<b>1 Die Entstehung von Perl</b>	<b>4</b>
1.1 Geschichte	4
1.2 Anwendungsgebiete	4
1.3 CPAN	4
1.4 parrot und die Zukunft von Perl	5
<b>2 Grundlagen</b>	<b>5</b>
2.1 Syntax - Beispiel: Hello World!	5
2.2 Perl Programmers Reference Guide	6
2.3 Kompilierung	6
2.3.1 Phase 1 - Compilation	6
2.3.2 Phase 2 - Code Generation	6
2.3.3 Phase 3 - Parse Tree Reconstruction	7
2.3.4 Phase 4 - Execution	7
2.4 Variablendeklaration und Variablenreichweite	7
2.5 Datentypen	8
2.5.1 Integer und Strings	8
2.5.2 Array und Hash	8
2.5.3 Referenzen	9
2.5.4 Quoting	9
2.6 Boolesche Ausdrücke	10
2.7 if, unless, else und elsif	10
2.8 Schleifen	11
2.8.1 Die while-Schleife	11
2.8.2 Die for-Schleife	11
2.8.3 Die foreach-Schleife	11
2.8.4 next und last	12
2.9 Einfache E/A	12
2.9.1 Interaktion mit STDOUT und STDERR	12
2.9.2 Öffnen von Dateien	12
2.10 Funktionen und Variablenreichweite	13
2.11 Pakete und Module	14
2.11.1 Pakete	14
2.11.2 Nutzen von Modulen	14
2.11.3 Aufbau eines Moduls	15
2.12 Reguläre Ausdrücke	15
2.13 Interprozess-Kommunikation	16
2.13.1 Signals	16
2.14 Threads	16
2.15 Perl Debugger	17
<b>3 LAMP</b>	<b>17</b>
3.1 perl::CGI	17
3.1.1 Prozedural vs. Objektorientiert	18
3.1.2 Parameter	18
3.1.3 Kompatibilität und Konformität	19
3.1.4 perl::CGI und CSS	19
3.1.5 perl::CGI und mod.perl	20

---

3.2	DBD::mysql	20
3.2.1	Verbindung	20
3.2.2	Datenbankanfragen	21
3.2.3	Fehlerbehandlung	21
4	Zusammenfassung	21

# 1 Die Entstehung von Perl

Perl ist älter, als manche Konzepte und Funktionsweisen errahnen lassen. Die Geburtsstunde von Perl war im Jahre 1987 und ging von Larry Wall<sup>1</sup> aus. Wall arbeitete zu dieser Zeit für die Firma Unisys und hatte die Aufgabe Logdateien eines Netzwerkes zu analysieren, welches sich über die gesamte USA erstreckte und die Logdateien über selbiges verteilte. Da es zu dieser Zeit kein Tool unter UNIX gab, welches dieser Aufgabe gewachsen war, entwickelte er Perl - die *Practical Extraction and Report Language* und stellte sie als frei verfügbare Software am 18.12.1987 ins Usenet.

## 1.1 Geschichte

Nach dem 18.12.1987 wurde Perl stetig erweitert und es entwickelte sich die Perl Community, die heute eine der größten und bekanntesten in der Open Source Gemeinde ist. Ein großer Boom konnte verzeichnet werden, als Mitte der 90er Jahre das Internet schlagartig weltweit expandierte und Perl sich für die Entwicklung von Internetangeboten, insbesondere *CGI* (siehe Abschnitt 3.1), als geeignet herausstellte. Mit der Version 5 wurden dann auch Konzepte wie Pakete (Abschnitt 2.11.1), Module (Abschnitt 2.11.1) und objektorientierte Programmierung zu Perl hinzugefügt. Derzeit aktuell ist die Version 5.8.2. Larry Wall hat sich hierbei aus dem Perl Alltag zurückgezogen und arbeitet eher an globalen Konzepten als an Bugfixes, um Perl für die Zukunft (siehe Abschnitt 1.4) zu rüsten.

## 1.2 Anwendungsgebiete

Perl wurde entwickelt, um das Arbeiten unter *UNIX* zu erleichtern. Zunächst war es dazu gedacht Textdateien zu parsen, zu bearbeiten und für das menschliche Auge lesbar zu machen. Nach und nach hat sich Perl jedoch zu einer kompletten Programmiersprache, auch im Sinne der theoretischen Informatik, entwickelt. Jedoch im Vergleich zu anderen Sprachen mit dem Vorteil, dass leichte Dinge leicht und schwere Dinge möglich sind. Desweiteren ist Perl mittlerweile auf allen wichtigen Systemen verfügbar und somit portabler als die meisten anderen Sprachen.

## 1.3 CPAN

Das *CPAN - Comprehensive Perl Archive Network*<sup>2</sup> archiviert Perl Module (siehe Abschnitt 2.11.2). Desweiteren ermöglicht es auf einfache Art und Weise, Perl Module zu installieren und aktualisieren. Dies ist insofern hilfreich, da es eine große Anzahl fertiger Perl Module zu den verschiedensten Aufgabenbereichen gibt. Oftmals ist es besser, auf ein fertiges Modul zurück zu greifen, also einen ähnlichen Code selbst zu schreiben.

<sup>1</sup>Auch bekannt durch das *patch* Programm

<sup>2</sup>siehe <<http://www.cpan.org>>

## 1.4 parrot und die Zukunft von Perl

Zwar ist immernoch die Version 5.8.2 die aktuelle Version, doch seit einiger Zeit konzentrieren sich die Entwickler, allen voran Larry Wall, darauf, eine neue Version von Perl zu schaffen. Hierbei geht es nicht nur um neue Funktionen sondern um einen kompletten Rewrite. Es ist vorallem bemerkenswert, wieviel Arbeit auf die Planung (Beginn: 18.07.2000) verwendet wurde und wie stark die Perl Community in selbige eingebunden wurde. Hierbei ist das Konzept ein ganz anderes. Nicht wie bisher wird Perl portiert werden müssen, sondern zwischen Betriebssystem und Perl ist *parrot* gerutscht. *parrot* ist ein auf Registern basierender, objektorientierter, bytecode gesteuerter, asynchroner Interpreter. Desweiteren unterstützt er auch die JIT<sup>3</sup> Ausführung. Somit muß lediglich eine *parrot* Implementation vorhanden sein, damit man Perl 6 auf der entsprechenden Plattform benutzen kann. Die Entwickler wünschen sich mit diesem Schritt aber auch mit dem allgemeinden Konzept Perl für die nächsten 20 Jahre zu rüsten und einige Fehler, ob konzeptual oder funktional, zu beheben. Sicherlich wird noch einige Zeit vergehen, bis die erste Beta-Version von Perl 6 zu beziehen ist, trotz allem wird an der Perl 5 Codebase weiterentwickelt und Fehler behoben. Nach vielen Versuchen sind z.B. die Perl Threads seit Version 5.8.0 offiziell für Produktionsumgebungen freigegeben.

## 2 Grundlagen

Das Folgende Kapitel beschäftigt sich mit der Perl Syntax, den Variablen und Datentypen sowie einigen tiefergehenden Konzepten.

### 2.1 Syntax - Beispiel: Hello World!

Hier ein einfaches *Hello World!* Programm:

```
#!/usr/bin/perl
# Hello World Programm

print "Hello World!\n";
```

Mit der ersten Zeile legt man den Perl Interpreter für Linux/Unix Systeme fest, welcher meistens unter */usr/bin/perl* zu finden ist. Unter Windows kann man die Dateieindung *.pl* mit der Datei *perl.exe* assoziieren, um Perl Programme direkt zu starten.

Kommentare werden, wie in der zweiten Zeile, durch eine Raute *#* makiert und gehen bis zum Ende der Zeile.

Die vierte Zeile sorgt dafür, daß *Hello World!* über die Standardausgabe (siehe Abschnitt 2.9) ausgegeben wird. Desweiteren ist zu erkennen, dass Befehle durch ein Semikolon getrennt werden.

Sollte Perl nicht installiert sein, kann es von <http://www.perl.org> bezogen werden. Aktuell ist die Version 5.8.2. Die meisten Linuxdistributionen enthalten schon vorgefertigte Pakete. Perl ist auch für Windows erhältlich.

Allgemein sind folgende Endungen typisch für Perl :

---

<sup>3</sup>Just-In-Time

<code>.pl</code>	PerlSkript
<code>.pm</code>	PerlModul

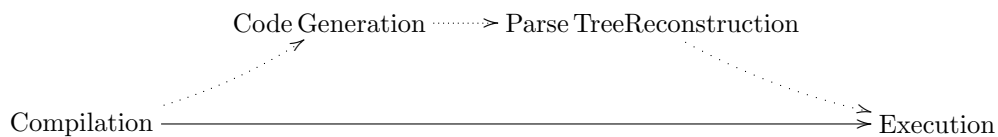
## 2.2 Perl Programmers Reference Guide

Der *Perl Programmers Reference Guide* kurz *perldoc* ist eine Möglichkeit, ohne teure Bücher oder das Internet zu erfahren, was eine Funktion, ein Keyword oder ein Modul macht.

```
awjb@75# perldoc -f print
print FILEHANDLE LIST
print LIST
print Prints a string or a list of strings. Returns true if successful. FILEHANDLE may be a scalar variable name, in which case the variable contains the name of or a reference to the filehandle, thus introducing one level of indirection.[...]
```

## 2.3 Kompilierung

Im Vergleich zu anderen Sprachen wie *C/C++* werden Perl Programme nicht kompiliert ausgeliefert sondern bei jedem Programmaufruf neu kompiliert. Dies geschieht in mehreren Schritten.



### 2.3.1 Phase 1 - Compilation

In der *Compilation Phase* wird der so genannte *Parse Tree* erzeugt, welcher im wesentlichen eine Baumstruktur ist, die den geparsen Programmcode enthält. In dieser Phase werden unter anderem auch alle regulären Ausdrücke kompiliert sowie Konstanten (vor-)ausgewertet. Sollten sich `INIT` oder `END` Blöcke im Sourcecode befinden werden diese zunächst ignoriert und später entsprechend ausgewertet. Sind `CHECK` Blöcke vorhanden, so werden diese nach dem *LIFO*<sup>4</sup>-Prinzip in Phase 2 ausgewertet.

### 2.3.2 Phase 2 - Code Generation

Diese Phase ist optional und wird nur dann durchlaufen, wenn im Sourcecode während Phase 1 `CHECK` Blöcke gefunden werden. Werden `CHECK` Blöcke gefunden so bedeutet dies, dass ein *Code Generator* genutzt werden soll. Der *Code Generator* stellt drei Möglichkeiten zur Verfügung: `B::Bytecode`, `B::C` und `B::CC`. Die erste Möglichkeit wandelt den Inhalt des *Parse Tree* in Perl Bytecode<sup>5</sup> um. Die beiden anderen Möglichkeiten erzeugen *C/C++* Sourcecode, der in ein ausführbares Programm umgewandelt werden kann. Sollte dies der Fall sein,

<sup>4</sup>last in,first out

<sup>5</sup>ähnlich *Java*

wird Phase 3 übersprungen. Wird Bytecode erzeugt, wird bei Phase 3 weitergemacht.

### 2.3.3 Phase 3 - Parse Tree Reconstruction

In dieser Phase wird der *Parse Tree* aus dem in Phase 2 erzeugten Bytecode rekonstruiert da eine direkte Ausführung des Bytecode zu langsam wäre.

### 2.3.4 Phase 4 - Execution

In dieser Phase wird das Programm ausgeführt, je nach dem aus dem *Parse Tree* oder als *C/C++* Programm mit integriertem Perl Interpreter. In dieser Phase werden nach dem *FIFO*<sup>6</sup>-Prinzip INIT Blöcke abgearbeitet und das Programm ausgeführt.

## 2.4 Variablendeklaration und Variablenreichweite

In Perl können, sofern nicht `use strict`<sup>7</sup> verwendet wird, Variablen einfach benutzt werden, ohne vorher eine Deklaration / Initialisierung vorgenommen zu haben.

```
$foo = "sometext";
```

Hierbei wird der Variablen `$foo` der Text `sometext` zugewiesen. Ähnlich zu den bekannten Keywords *public, private, protected* aus *C++* gibt es in Perl die Keywords `our`, `local`, `my`. Mit diesen Keywords ist es möglich zu entscheiden, ob

`our` der Variablenname auf das Scope festgelegt wird

`local` der Wert auf das Scope festgelegt wird

`my` Wert und Variablenname auf das Scope festgelegt werden.

#### Beispiel:

```
my $foo = 1;
```

```
if($foo)
{
    local $foo = 2;
    print "1: $foo\n";
}
```

```
print "2: $foo\n";
```

```
=> 1: 2
```

```
=> 2: 1
```

Wie man an diesem Beispiel sieht, muß man sehr genau nachdenken, welches Keyword man nutzt, denn es kann leicht zu Verwechslungen und somit zu unvorhergesehenen Verhalten im Programm kommen.

<sup>6</sup>first in, first out

<sup>7</sup>siehe Abschnitt 2.15

## 2.5 Datentypen

Wie in anderen Programmiersprachen auch, stellt Perl einige grundlegende und einige weiterführende Datentypen zur Verfügung. Hierbei legt Perl zunächst nicht, wie bei *C* den Type *char, int, float, ...* fest, sondern macht dies aus dem Kontext heraus.

Im folgenden wird an Codebeispielen der Umgang und die Verwendung gezeigt:

### 2.5.1 Integer und Strings

Zunächst die Deklaration verschiedener Variablen:

```
my $number = 1;
my $char = 'f';
my $string = 'oo';
```

Natürlich unterstützt Perl die Grundrechenarten wie  $+, *, -, \dots$

```
my $number = 2;
$number = $number - 2;
$number += 1;
print $number;
=> 1
```

Mit dem Operator `.=` können Strings konkateniert werden:

```
my $string = "ab";
my $char = "c";
$string .= $char;
print $string;
=> abc
```

### 2.5.2 Array und Hash

Perl unterstützt als erweiterte Datentypen Array und Hash. Diese beiden Datentypen sind dynamisch, das heißt sie haben keine vorgegebene Länge und werden je nach Bedarf vergrößert. Desweiteren gibt es auch hier keine feste Typisierung. Daher können beliebige Daten und Datentypen als Wert verwendet werden. Wenn notwendig übernimmt Perl dabei die (De-)Referenzierung<sup>8</sup>.

```
my @array;

$array[0] = 1;
$array[1] = 'a';
print @array;
=> 1 a
```

Ein Array in Perl beginnt immer an der Position 0. Wird der ganze Array angesprochen, so wird ein `@` verwendet. Wird auf Werte im Array zugegriffen so wird `$` verwendet und in `[ ]` die Position angegeben. Um die aktuelle Länge eines Array zu erfragen, genügt folgender Befehl:

<sup>8</sup>Mehr zu Referenzen im Abschnitt [2.5.3](#)

```
my $length = scalar(@array);
print "Laenge: $length\n";
=> Laenge: 2
```

Ähnlich wie bei einem Array wird der Zugriff auf einen Hash über `$` gesteuert. Hierbei wird der Hashschlüssel in `{ }` hinter den Bezeichner des Hash geschrieben. Wird der ganze Hash angesprochen wird ein `%` vor den Bezeichner geschrieben.

```
my %hash;
$hash{1} = 1;
$hash{abc} = 23;
$hash{5} = "foo";
$hash{a} = @array;
print $hash{abc};
=> 23
```

### 2.5.3 Referenzen

Ähnlich wie in *Java* gibt es in Perl so genannte Referenzen. Man kann sie zum (de-)referenzieren nutzen<sup>9</sup>. Um zu erfragen, um welchen Datentype es sich bei einer Referenz handelt, kann der Befehl `ref` genutzt werden.

```
my $reference = \@array;

if (ref($ref) eq 'ARRAY')
{
    print "Is array reference.\n";
}
else
{
    print "Is not a reference.\n";
}
=> Is array reference.
```

### 2.5.4 Quoting

Perl unterstützt verschiedene Formen des Quotings. Wird z.B. ein String in Singlequotes `'` gesetzt, so wird sein Inhalt nicht ausgewertet. Bei Doublequotes `"` wird jedoch der Inhalt vor der Zuweisung ausgewertet und der entsprechend ausgewertete Wert zugewiesen.

```
my $foo = 'abc';
my $sq = '$foo';
my $dq = "$foo";
print "$sq\n";
print "$dq\n";
=> $foo
=> abc
```

---

<sup>9</sup>vergl.: *C/C++* \*,\$

Das Quoting mit Anführungszeichen ist jedoch bei der Verwendung von *HTML*<sup>10</sup> sehr umständlich, da zunächst das entsprechende Anführungszeichen mit einem Backslash versehen werden muß, da es sonst den Anfang oder das Ende des Strings bewirken würden.

```
my $link = "<a href="http://foo.org">A nice link to foo</a>";
```

Dieses Beispiel würde zu einem Syntaxfehler führen, da der String zwischendurch durch Doublequotes beendet und neu angefangen wurde.

```
my $link = "<a href=\"http://foo.org\">A nice link to foo</a>";
print $link;
=> <a href="http://foo.org">A nice link to foo</a>
```

Dieses Beispiel würde funktionieren, jedoch zeigt sich schnell, dass dies sehr umständlich für den täglichen Bedarf ist.

Aus diesem Grund stellt Perl eine praktische Quoting-Funktion zur Verfügung. Sie erlaubt es, auf einfache Weise andere Zeichen als Stringbegrenzung zu nutzen.

```
my $link = qq [<a href="http://foo.org">A nice link to foo</a>];
print $link;
=> <a href="http://foo.org">A nice link to foo</a>
```

Eine ähnliche Funktion ist `qw` im Bezug auf Listen von Strings:

```
my @foo = ("first", "second", "third");
my @bar = qw( first second third );
print "@foo\n";
print "@bar\n";
=> first second third
=> first second third
```

## 2.6 Boolsche Ausdrücke

Perl kennt verschiedene Boolsche Ausdrücke und unterscheidet in verschiedene Wertigkeiten. Im allgemeinen gilt: Buchstaben haben eine geringere Priorität als Zeichen. Dies gilt jedoch nur bei Verknüpfungen nicht bei Vergleichen.

### Beispiele

Operator1	Operator2	Bedeutung
&&	<i>and</i>	Verknuepfung
==	<i>eq</i>	Gleichheit
!=	<i>ne</i>	Ungleichheit
!	<i>not</i>	Negation

## 2.7 if, unless, else und elsif

Wie in z.B. *C/C++* unterstützt Perl die Keywords `if` und `else`. Hinzu kommt das Keyword `elsif` welches eine Kurzform für `else if` ist und das Keyword `unless` welches eine Kurzform für `if(!expr)` ist.

### Beispiel:

<sup>10</sup>HTML - Hyper Text Markup Language

```
my $i = 1;
if($i == 2)
{
    print "i: 2\n";
}
elsif($i == 1)
{
    print "i: 1\n";
}
else
{
    print "i: $i\n";
}
=> i: 1
```

## 2.8 Schleifen

Dieser Abschnitt geht auf verschiedene Typen von Schleifen in Perl ein.

### 2.8.1 Die while-Schleife

Die `while`-Schleife besteht aus dem Keyword `while`, einer Bedingung in runden Klammern und dem Sourcecode der ausgeführt werden soll, solange die Bedingung wahr ist.

**Beispiel:**

```
my @foo = (1,2,3,4,5,6,7,8,9,0);
my $length = scalar(@foo)-1;

while($length != 0)
{
    $foo[$length] = 0;
    $length--;
}
```

### 2.8.2 Die for-Schleife

Die `for`-Schleife unterscheidet sich von der `while`-Schleife, dadurch dass die Bedingung durch eine Zählvariable ausgedrückt wird.

```
for(my $i = 0; $i<6; $i++)
{
    print "$i...";
}
=> 0...1...2...3...4...5...
```

### 2.8.3 Die foreach-Schleife

Die `foreach`-Schleife ist aus Sprachen wie *C/C++* oder *Java* nicht bekannt. Sie eignet sich besonders zum Durchlaufen von Hashschlüsseln etc.

```
foreach $key (sort(keys %hash))
{
    #do something
}
```

#### 2.8.4 next und last

Unabdingbar für das Arbeiten mit Schleifen ist die Möglichkeit aus selbigen herauszuspringen oder frühzeitig den aktuellen Durchlauf zu beenden. Hierzu gibt es in Perl die Keywords `last` und `next`.

```
for(my $i=0;$i<100;$i++)
{
    if($i<10)
    {
        next;
    }
    elsif ($i == 23)
    {
        last;
    }
}
```

## 2.9 Einfache E/A

Aufgrund der Entstehungsgeschichte erlaubt Perl auf einfache Weise E/A mit dem System. Hierzu werden verschiedene Methoden zur Verfügung gestellt.

### 2.9.1 Interaktion mit STDOUT und STDERR

Wie in den meisten Sprachen üblich, stellt Perl selber einige vordefinierte Datei-Handle zur Verfügung. Dieses Beispiel zeigt den Umgang von `STDOUT` und `STDERR` mit `print`.<sup>11</sup>

```
print STDOUT "Hello World!\n";
print STDERR "Could not do foo...!\n";
```

Dieses Beispiel würde auf der Standardausgabe des Systems die Meldung *Hallo Welt!* ausgeben. Der zweite Befehl würde eine Fehlermeldung über die Fehlerausgabe des Systems ausgeben...

### 2.9.2 Öffnen von Dateien

Natürlich kann man mit Perl auch Dateien öffnen. Hierzu werden verschiedene Modi<sup>12</sup> bereitgestellt:

#### Beispiele:

<sup>11</sup>Wird bei `print` kein Datei-Handle angegeben, so wird `STDOUT` genutzt

<sup>12</sup>Vergleiche hierzu die Dateimodi im *ANSI C* Standard *ANSI X3.159-1989*

Zeichen	Modus
>	schreiben
<	lesen
>>	hinzufuegen
+<	lesen und schreiben
	Prozess Pipe

Diese können in Kombination mit z.B. der `open()` Funktion dazu genutzt werden um Dateien zu öffnen. Aber Perl kann nicht nur aus Dateien lesen oder in selbige schreiben, sondern auch direkt von Prozessen lesen und an Prozesse Output weitergeben. Das Filehandle kann mit der Funktion `close()` wieder geschlossen werden. Hierbei sollten Großbuchstaben verwendet werden, damit nicht aus versehen Keywords und Filehandle verwechselt werden. Wichtig ist, dass das Filehandle, wie im unteren Beispiel, von der `open()`-Funktion erzeugt wird. Somit darf es vorher nicht deklariert werden.

```
open(F00,">","my.bar");
print F00 "Nice text...\n";
close F00;
```

Alternativ kann auch die Syntax `open(F00,»my.bar");` verwendet werden. Da Perl auch *Unicode* unterstützt, kann man beim Öffnen des Filehandle auch angeben, was für ein Input gelesen bzw. geschrieben werden soll.

```
open(F00,"<:utf8","my.bar");
while(my $line = <F00>)
{
    print "$line";
}
close F00;
```

## 2.10 Funktionen und Variablenreichweite

Genau wie in *C/C++* oder ähnlichen Sprachen ist es in Perl möglich, Unterrou-tinen/Funktionen zu definieren. Hierzu wird das `sub`-Keyword genutzt. Auch hier können Parameter übergeben und entsprechend Werte zurückgegeben werden. Desweiteren können Funktionen auch einfach referenziert werden. Hierbei sind die übergebenen Werte im Array `@_` zu finden.

```
if(isfoobar($foo))
{
    #do something
}

sub isfoobar
{
    my $foo = $_[0];
    my $bar = 1;
    if($foo > $bar)
    {
        return 0;
    }
}
```

```
    }  
    return 1;  
}
```

## 2.11 Pakete und Module

Um effektiver mit Perl arbeiten zu können gibt es Pakete und Module. Diese sollen in den folgenden Abschnitten vorgestellt werden.

### 2.11.1 Pakete

Pakete sind in Perl gleichbedeutend mit Namespace. Pakete kommen dann ins Spiel, wenn es darum geht, Sourcecode übersichtlich zu gestalten und (mit Konzepten aus Abschnitt 2.10) Konflikten zwischen verschiedenen Teilen des Codes schon von Anfang an zu umgehen. Hierzu erzeugt Perl sogenannte *symbol table*. Eine Symboltabelle ist ein Hash mit den Werten des Packetes. Hierbei wird die globale Symboltabelle mit `%main::` bzw. der Abkürzung `%::` angegeben.

```
local $foo="hello";  
  
bar();  
  
sub bar  
{  
    local *foobar = *::foo;  
    #do stuff...  
}
```

Zu beachten ist, dass `%main::` neben allen anderen Symboltabellen auch sich selbst als `%main::main::...::` referenziert.

### 2.11.2 Nutzen von Modulen

Ein weiterer Schritt, um Sourcecode übersichtlicher zu gestalten sind die Module. Wie in Abschnitt 1.3 angesprochen, gibt es ähnlich *Java* schon eine große Sammlung von Modulen. Module enthalten Perl Code. Zu beachten ist, dass nicht wie bei *C++* oder *Java* eine strikte Einhaltung des Variablenscopes sichergestellt wird.

Um Module zu laden, gibt es zwei Möglichkeiten:

1. `use foomodule`  
Mit dem Keyword `use` wird das Paket eingebunden und alle vom Modul als Standard gesetzten Funktionen geladen. Dies geschieht während der Kompilierphase
2. `require foomodule`  
Mit dem Keyword `require` werden auch alle als Standard gesetzten Funktionen geladen, jedoch erst nach der Kompilierphase.

Daher sollte immer `use` genutzt werden. Werden zusätzliche Parameter an das Modul übergeben, so können gezielt Funktionen importiert werden.

```
use foomodule qw(foo bar);
```

Dies würde die Funktionen *foo* und *bar* aus dem Modul *foomodule* importieren.

### 2.11.3 Aufbau eines Moduls

Damit ein Perl Programm ein Modul benutzen kann, muß dieses gewisse Voraussetzungen erfüllen. Wir wollen an einem Beispiel den groben Aufbau zeigen:

```
package Foobar;
```

Diese Zeile erzeugt das Modul *Foobar*.

```
require Exporter;
our @ISA = qw(Exporter);
```

Nun wird das Modul *Exporter* geladen, damit *Foobar* Funktionen exportieren kann.

```
our @EXPORT = qw(foo);
our @EXPORT_OK = qw(bar);
```

Die erste Zeile legt fest, was exportiert wird, wenn das Modul mit `use Foobar` in einem Programm geladen wird. Die zweite Zeile legt fest, was auf besondere Anforderung hin exportiert werden darf.

```
our $VERSION = '$Id:$';
```

Hier wird die Version des Modules gespeichert damit bei der Verwendung überprüft werden kann, ob es sich um die richtig Version mit allen Features handelt.

```
my $bar = 'stuff';
```

```
sub foo
{
    # do foo
}
1;
```

Nun folgt der eigentliche Sourcecode des Moduls. Wichtig für den Compiler ist, dass am Ende eine `1;` steht.

## 2.12 Reguläre Ausdrücke

In Perl ist es auf einfache Weise möglich *regular expressions* (reguläre Ausdrücke) zu benutzen. Anders als z.B. in *C* müssen in Perl die regulären Ausdrücke nicht extra vor dem Gebrauch kompiliert werden. Auch das *pattern matching* geht wesentlich einfacher, als mit *regex* in *C*. Je nach Anwendung stellt Perl verschiedene Möglichkeiten zur Verfügung. Zum reinen *pattern matching* kann das `m//` genutzt werden. Um anhand von regulären Ausdrücken Teile eines Strings zu substituieren, nutzt man `s///`. Um z.B. die Groß- und Kleinschreibung zu ändern, kann `tr///` genutzt werden.

**Beispiel:**

```
if ( $foobar =~ m/foo/ )
{
    #do foo...
}
```

Wird in der Variablen `$foobar` der String `foo` gefunden wird der Sourcecode in der Schleife ausgeführt.

```
my $foobar = "foo";
$foobar =~ s/foo/bar/;
print "$foobar\n";
=> bar
```

```
$foobar =~ tr/a-z/A-Z/;
print "$foobar\n";
=> BAR
```

```
$foobar =~ m/([A-B]*)/;
print "Match was $1\n";
=> Match was BA
```

## 2.13 Interprozess-Kommunikation

Perl kennt viele verschiedene Arten der Interprozesskommunikation. *Signals*, *Sockets*, *System V IPC* und *Pipes*, um nur einige zu nennen. Während *Pipes* schon in Abschnitt 2.9.2 beschrieben wurden, sollen hier nur das Signalhandling von Perl weiter gezeigt werden.

### 2.13.1 Signals

Wie sicher aus der *Unix/Linux* Umgebung bekannt, unterstützt das Betriebssystem verschiedene Signale<sup>13</sup>, die es an Prozesse senden kann. Perl stellt hierfür den Hash `%SIG` zur Verfügung.

## 2.14 Threads

Seit Perl Version 5.8.0 gibt es Threading. Hier soll gezeigt werden, wie man mit Threads arbeitet. Wichtig ist, daß das Modul *Thread* eingebunden wird. Danach stellt selbiges die Methoden zum threading zur Verfügung.

### Beispiel:

```
use Thread;

$foothread = Thread->new( sub { foo($bar) });

my $returnvalue = $foothread->join();
```

In diesem Beispiel wird zunächst ein Thread mit `Thread->new()` gestartet und dann der Rückgabewert mit `$foothread->join()` abgerufen. Zu beachten ist hier noch, dass `$foothread->join()` einen Rückgabewert liefert, wenn der

<sup>13</sup>Die genauen Signalwerte und Signaltypen sind in den Systembibliotheken zu finden

Thread `$foothread` fertig ist, sonst den aufrufenden Thread so lange suspendiert, bis `$foothread` fertig ist.

## 2.15 Perl Debugger

Funktioniert ein Perl Programm nicht, oder nicht so wie erwartet, gibt es mehrere Möglichkeiten Fehler zu finden. Um einfache Fehler zu finden, sollte man generell Perl mit der Option `-w` aufrufen. Dies gibt Warnungen aus. Eine weitere Möglichkeit besteht in der Benutzung des Paketes *strict*. Durch ein `use strict;` am Anfang des Sourcecodes wird geprüft, daß z.B. Variablen deklariert werden. Sollte auch das noch nicht den gewünschten Erfolg liefern, so gibt es weiterhin die Möglichkeit, den wirklichen Perl Debugger zu starten. Dies geschieht mit der Option `-d`. Wird Perl mit dieser Option gestartet, so befindet man sich in der Debugger Umgebung ähnlich der des *gdb*<sup>14</sup>. So kann man z.B. mit `b foofunction` Breakpoints setzen, mit `c` den Code bis zum nächsten Breakpoint oder Fehler weiterlaufen lassen. `w` gibt ein Stück des Sourcecode aus und `T` einen *stack backtrace*.

## 3 LAMP

Seit einiger Zeit wird im Internet auf vielen Servern eine Kombination von Linux, Apache, MySQL (Abschnitt 3.2) und Perl (Abschnitt 3.1) verwendet. Aus dieser Kombination ging der Begriff *LAMP* hervor. Diese Kombination ist daher so günstig, da alle Programme frei verfügbar sind und sich sowohl im Bereich Performance als auch Stabilität gegenüber ihren kommerziellen Gegnern behaupten können. Im folgenden nun einige Beispiele zur Konfiguration, Handhabung und Vorgehensweise.

### 3.1 perl::CGI

Perl kann das *CGI*<sup>15</sup> dazu nutzen, dynamische, interaktive Webseiten zu gestalten. Eines der unzähligen *CGI* Module ist *CGI.pm* und kann vom *CPAN* (siehe Abschnitt 1.3) bezogen werden.

```
use CGI;  
$cgi = new CGI;
```

Zunächst wird das Modul *CGI* geladen und ein neues CGI-Objekt angelegt.

```
print $cgi->header();  
print $cgi->start_html(-title=>'foopage');
```

Mit der Funktion `header()` wird zunächst der *Content-Type* ausgegeben und mit `start_html` ein kompletter *HTML* Kopf.

```
print $cgi->h1('This is foopage');  
print $cgi->end_html();
```

---

<sup>14</sup>`gdb` - the gnu debugger <[www.gnu.org](http://www.gnu.org)>

<sup>15</sup>Common Gateway Interface

Mit `h1()` wird eine H1 HTML Überschrift ausgegeben und mit `end_html` das schließende `</body>` und `</html>`.

Ähnlich zu den hier aufgeführten Funktion `h1` gibt es eigentlich alle aus *HTML* bekannten Tags auch als Funktionen in `perl::CGI`. Eine der Ausnahmen ist `<tr>` da dies ja in Perl schon belegt ist (siehe Abschnitt 2.12). Hierfür wird dann Wahlweise `Tr` oder `TR` genutzt.

### 3.1.1 Prozedural vs. Objektorientiert

Beispiel 3.1 scheint auf den ersten Blick recht kompliziert, weil immer mit dem Object `$cgi` gearbeitet werden muß. Hierbei spricht man auch vom objektorientierten *CGI*. Deutlich einfacher scheint da prozedurales *CGI* zu benutzen. Hierzu betrachten wir das Beispiel aus Abschnitt 3.1 in prozeduralem *CGI*.

```
use CGI qw(:standard);

print header();
print start_html(-title=>'foopage');

print h1('This is foopage');

print end_html();
```

Sicherlich bleibt jedem überlassen, welche der beiden Möglichkeiten er nutzt. In jedem Fall sollt auf Konsistenz geachtet werden, damit ein Projekt übersichtlich bleibt. In den weiteren Abschnitten wird die Schreibweise aus Abschnitt 3.1 verwendet, um deutlicher unterscheiden zu können.

### 3.1.2 Parameter

Natürlich kann man mit *CGI* auch auf GET/POST Parameter zugreifen. Hierzu wird die Funktion `param()` bei POST bzw. `url_param()` bei GET verwendet.

```
# fetching all parameters
@foovalues = $cgi->param();

# fetching parameter foo
$foovalue = $cgi->param('foo');
```

Desweiteren können Parameter besetzt und gelöscht werden (nur POST):

```
# delete parameter foo
$cgi->delete('foo');

# delete all parameters
$cgi->delete_all();
```

Wem dies zu unpraktisch ist, der hat die Möglichkeit alle Parameter in einen eigenen Namespace zu importieren, um dann einen direkten Zugriff auf selbige zu ermöglichen. Wichtig ist jedoch, nicht in den *main* Namespace zu importieren, da hierdurch nicht unerhebliche Sicherheitslücken entstehen können.

```
# import all parameters to namespace bar
$cgi->import_names{'bar'};
```

```
# work with namespace bar
print "$bar::foo\n";
```

Um alle Parameter bzw. Forminformationen zu speichern gibt es in `perl::CGI` zwei Möglichkeiten:

```
# save to file
$cgi->save(FOOFILE);
```

```
# save to variable
my $foourl = $cgi->self_url;
```

### 3.1.3 Kompatibilität und Konformität

Werden, z.B. in *HTML*-Formularen, Strings benutzt, so werden sie Standardmässig so formatiert, dass der Programmierer sie nicht mehr escapen muß. Diese Funktion, dass so genannte *autoEscape*, kann aus- bzw. eingeschaltet werden:

```
# disable autoEscape
$cgi->autoEscape(undef);
```

```
#enable autoEscape
$cgi->autoEscape('true');
```

Desweiteren gibt es einige Funktionen, mit denen man gezielt Strings in *URL* bzw. *HTML* konforme Strings umwandeln kann (und natürlich umgekehrt). Diese Funktionen müssen beim Laden des *CGI* Moduls extra geladen werden:

```
use CGI qw(escape unescape escapeHTML unescapeHTML);
```

```
my $escaped_url = escape('$somesstring with /\<>*$^&$$#@! chars$');
print "$escaped_url\n";
=> %24somesstring%20with%20%2F%5C%3C%3E%2A%25%24%5E%26%24%23%40%21%20chars%24
```

```
print "unescape($escaped_string)\n";
=> $somesstring with /\<>*$^&$$#@! chars$
```

```
my $escaped_html = escapeHTML('now we <don't> like stuff <like> this <html>');
print "$escaped_html\n";
=> now we &lt;don't&gt; like stuff &lt;like&gt; this &lt;html&gt;
```

```
print unescapeHTML($escaped_html);
=> now we <don't> like stuff <like> this <html>
```

### 3.1.4 perl::CGI und CSS

`perl::CGI` bietet auch Support für *CSS*<sup>16</sup>. Hierzu muß jedoch der Parameter `:html3` beim Laden des Modules übergeben werden (bei neueren Versionen von

<sup>16</sup>Cascading Style Sheets

*CGI.pm* entfällt dies). Dann können zusätzlich zum Attribut *-class* auch *-style* und das Element *span* verwendet werden.

Möchte man im Quelltext des Programms ein Stylesheet definieren, kann dies wie folgt mit einem so genannten *here-doc* gemacht werden. Dieses *here-doc* wird mit `<<` und einem Namen, in diesem Fall `END` markiert. Das bedeutet, dass bis zur Zeile mit `END` alles in die Variable `$style` geschrieben wird.

```
my $style = <<"END";
>!--
/* Style Sheet Definitions... */
-->
END
```

### 3.1.5 perl::CGI und mod\_perl

Wenn `perl::CGI` zusammen mit *mod\_perl* verwendet wird, so muß beim Laden folgendes beachtet werden:

Wird Perl in einer Version kleiner 5.003\_93 verwendet, muß

```
use CGI::Apache
```

statt

```
use CGI;
```

verwendet werden. Bei allen neueren Versionen sollten die Skripte wie gewohnt funktionieren. Nach dem dem Laden des *CGI*-Moduls sollte folgender Befehl im Sourcecode stehen.

```
CGI->compile(':all');
```

Hiermit werden alle `perl::CGI` Methoden einmal kompiliert und müssen aufgrund der besonderen Implementation von *mod\_perl* nicht wieder kompiliert werden.

## 3.2 DBD::mysql

*MySQL* ist eine Datenbank, die von <http://mysql.com> bezogen werden kann. Sie ist unter der *GPL/LGPL*<sup>17</sup> verfügbar.<sup>18</sup> Derzeit aktuell ist die Version *4.0.16*. Um *MySQL* unter Perl nutzen zu können, muß allerdings ein entsprechendes Modul installiert sein. Das Modul `DBD::mysql` kann über das *CPAN* (siehe Abschnitt 1.3) bezogen werden.

### 3.2.1 Verbindung

Zunächst muß das Modul geladen und eine Verbindung zur Datenbank hergestellt werden:

```
use DBI;
my $db = DBI->connect("DBI:mysql:database=foo;host=localhost",
                    "bar", "password", { 'RaiseError' => 1 });
```

<sup>17</sup>General Public License/Lesser General Public License

<sup>18</sup>Das Lizenzmodell ist sowohl GPL/LGPL als auch Kommerziell (siehe *MySQL* Webseite)

Erst wird das Modul DBI geladen. Danach wird mit `DBI->connect` eine Verbindung zur Datenbank aufgebaut und in diesem Fall im Datenbank-Handle `$db` gespeichert.

Nun sollte die Verbindung zur Datenbank hergestellt sein. Alles weitere kann nun über den Datenbank-Handle `$db` erledigt werden. Nachdem dieser nicht mehr gebraucht wird, sollte er mit `$db->disconnect()` geschlossen werden.

### 3.2.2 Datenbankabfragen

Nachdem die Verbindung zur Datenbank hergestellt worden ist, können nun Anfragen an die Datenbank gestellt werden. Hierbei sollte unbedingt darauf geachtet werden, dass nicht einfach Werte aus den POST oder GET Variablen in die Datenbankabfragen geschrieben werden, sondern diese Strings z.B. mit der `quote()` Funktion des Datenbank-Handle bearbeitet werden, um Sicherheitsrisiken zu vermeiden.

```
my $result = $db->prepare("SELECT * from foobar where id=".
                          $db->quote($foobar));
$result->execute();
```

Zunächst wird die Anfrage vorbereitet und in einem Query-Handle (`$result`) gespeichert. Danach wird sie mit `$result->execute()` ausgeführt.

```
while(my $rowref = $result->fetchrow_hashref())
{
    print "Id: $rowref->{'id'} Name: $rowref->{'name'}\n";
}
$result->finish();
```

`$result->fetchrow_hashref()` liefert die gefundenen Ergebnisse zurück und mit `$result->finish()` wird der verbrauchte Speicher des Query-Handle wieder freigegeben.

Sollen keine Werte zurückgegeben werden, so benötigt man kein eigenes Query-Handle. Hier reicht etwa folgendes:

```
$db->do("insert into foo values(bar)");
```

### 3.2.3 Fehlerbehandlung

Natürlich ist es auch möglich, Fehler abzufangen und auszuwerten. Hierbei stellt `DBD::mysql` die Attribute `mysql_errno` und `mysql_error` zur Verfügung. Über sie kann die entsprechende Fehlernummer bzw. die Fehlermeldung abgefragt werden:

```
my $errno = $db->{'mysql_errno'};
my $errmsg = $db->{'mysql_error'};
```

## 4 Zusammenfassung

Perl gibt es nun schon seit vielen Jahren. Die Entwicklung an Version 6 und die gesetzten Ziele zeigen, dass Perl immer noch zu den beliebtesten und bekanntesten Sprachen gehört. Sicherlich hat man in Perl viele Möglichkeiten

und aufgrund der Vielseitigkeit bietet es auch auf vielen Gebieten adäquate Lösungskonzepte. Ein Vor- und Nachteil ist, dass durch die Möglichkeit ein Problem auf viele verschiedene Weisen lösen zu können eine Umstellung von anderen Programmiersprachen vermeintlich einfach, ein lesen von Sourcecode anderer Programmierer aber vermeintlich schwer sein kann. Trotzdem ist Perl für den Einsatz also Websprache im Vergleich zu anderen Sprachen immer noch eine sinnvolle Wahl.

## Literatur

- Wall, Christiansen, Orwant  
Programming Perl  
O'Reilly, 3rd Edition
- Randal, Sugalski, Tötsch  
Perl 6 Essentials  
O'Reilly, 1st Edition
- CGI.pm - a Perl5 CGI Library  
[ftp://ftp-genome.wi.mit.edu/pub/software/WWW/CGI/cgi\\_docs.html](ftp://ftp-genome.wi.mit.edu/pub/software/WWW/CGI/cgi_docs.html)
- DBD::mysql - MySQL driver for the Perl5 Database Interface (DBI)  
<http://theoryx5.uwinnipeg.ca/CPAN/data/DBD-mysql/DBD/mysql.html>