# Einführung in PERL

Stefan Bucsics

PERL (**P**ractical **E**xtraction and **R**eport **L**anguage) wurde im Dezember 1987 von Larry Wall als Erweiterung der UNIX-Shells konzipiert und ist seitdem frei verfügbar (http://www.perl.com/). Die im Dezember 1999 gültige Version ist 5.005_03. Mittlerweile gibt es unzählige ebenfalls frei verfügbare Perlmodule und Bibliotheken für alle Bereiche, die in der riesigen (760 MB Text!) Sammlung *Comprehensive Perl Archive Network* (CPAN) zusammengefasst sind
http://gd.tuwien.ac.at/languages/perl/CPAN/).

Die Stärken von Perl sind das Bearbeiten von großen Texten, das Manipulieren von Dateien und Prozessen und die Netzwerkanbindung - zum Beispiel zu Datenbanken oder als CGI-Script..

Wegen des großen Erfolges wurde Perl auch auf andere Plattformen portiert, wie zu *ActivePerl* unter Windows oder *MacPerl* in der Applewelt. Naturgemäß unterscheiden sich diese Versionen in den systemnahen Bereichen von der Originalversion unter Unix.

Die Sprachsyntax orientiert sich stark an C, bietet aber einige neue Variablentypen wie das assoziative Array. Groß- und Kleinschreibung wird unterschieden!

Die hier in Beispielen vorgestellten Sprachelemente beziehen sich auf die Unix-Version von Perl und zeigen nur einen kleinen Teil der Möglichkeiten auf. So wird zum Beispiel nicht auf die objektorientierte Programmierung oder Datenbankanbindung unter Perl eingegangen.

## Variablentypen

### Einfache Variablen
### erstes Zeichen des Namens ist '$'

Sie enthalten entweder eine Zahl, einen Text oder eine Referenz; eine genauere Typisierung ist nicht möglich.

```
$x = 3.14;  $y = 2;  $z = $x+$y;        $z hat den Wert 5.14
$Name = "Fritz Nobody";
$Text = "Der Name ist $Name";           ergibt "Der Name ist Fritz Nobody"
$Text = 'Der Name ist $Name';           ergibt "Der Name ist $Name"
```

### Arrays
### erstes Zeichen des Namens ist '@'

```
@Tina= (-1, 2, 3, 4.51);                ein Array mit 4 Zahlenwerten
@Otto = ("Mister", "Strong");           ein Array mit zwei Textwerten
@Mixed = (1, "zwei", $x);               jede einfache Konstante oder Variable ist als Element möglich
@erweitert = (@Mixed, 4, "last');       ergibt (1, "zwei", 3.14, 4, "last") verschachtelte (mehrdimensionale) Arrays werden in eindimensionale Arrays umgewandelt!
```

### Zugriff auf Arrays

```
$Name = $Otto[1];                       ergibt "Strong"; "$Otto[1]"steht für eine einfache Variable
$Otto[1] = "Super";
@Neu = @Tina;                           eine Kopie von @Tina
@Neu = @Tina[1,2];                      ergibt (2,3)
@Otto[0,1] = @Otto[1,0];                vertauscht die Reihenfolge der Elemente
($a, $b) = (1, 2);                      ergibt $a=1; $b=2;
($a, $b) = ($b, $a);                    vertauscht die Variableninhalte von $a und $b
push(@Neu, 4);                          hängt an @Neu das Element 4 an, dh.@Neu = (2,3,4);
$Last = pop(@Neu);                      entfernt in @Neu das letzte Element und speichert dieses in $Last
$First = shift(@Neu);                   entfernt in @Neu das erste Element und speichert dieses in $First
@Sortiert = sort(@Unsortiert);          sortiert die Elemente eines Arrays
@Verkehrt = reverse(@Array);            dreht die Reihenfolge der Elemente eines Arrays um
```

### Assoziative Arrays
### erstes Zeichen des Namens ist "%"

Ist ein Array bestehend aus Elementenpaaren. Das erste Element eines solchen Paares nennt man den "Schlüssel" und das zweite Element den "zugehörigen Wert".

```
$Lager{"Bananen"} = 13;
$Lager{"Aepfel"} = "keine";             %Lager = ("Bananen", 13, "Aepfel", "keine")
%Lager = ("Bananen", 13, "Aepfel",      ergibt dasselbe
"keine");
%Vorrat = %Lager;                       ergibt eine Kopie
```

### Zugriff auf assoziative Arrays

Die Werte werden mittels des Schlüssels angesprochen.

```
$Neu = $Lager{"Bananen"};               ergibt $Neu = 13
$Lager{"Aepfel"} = 4*3;                 ergibt ("Bananen", 13, "Aepfel", 12)
@Lagerliste = %Lager;                   ergibt LagerListe als normales Array
@Namen = keys(%Lager);                  @Namen ist ("Bananen", "Aepfel") oder ("Aepfel", "Bananen")
@Werte = values(%Lager);                @Werte ist (13, 12) oder (12, 13)
@Paar =each(%Lager);                    liefert ein Elementenpaar der assoziativen Liste # (zumeist in while-Schleifen verwendet)
delete $Lager{"Aepfel"};                %Lager besteht nur mehr aus einem Paar
```

## Spezielle Variablen

Perl definiert eine Vielzahl von Systemvariablen, die unterschiedliche Daten wie UserID, Perl-Versionsnummer, Trennzeichen und ähnliches enthalten. Hier die wichtigsten:

| | |
|---|---|
| `$_ oder @_` | **wird defaultmäßig von einigen Funktionen mit Werten versehen** |
| `@ARGV` | **enthält die Übergabeparameter der Kommandozeile** |
| `%ENV` | **enthält die Environmentvariablen des Servers** |
| `@INC` | **enthält die Pfade zu den Bibliotheksdateien und Modulen** |

## Geltungsbereich von Variablen

Grundsätzlich sind normale Variablendeklarationen innerhalb eines „package" gültig. Dieses ist ein Codebereich, der durch die Zeile eröffnet wird:

```
package Paketname;
```

Der package-Bereich gilt solange, bis ein anderes package deklariert wird! Wurde das andere package aber schon einmal vorher deklariert, so gelten die alten Variablen mit den alten Werten wieder.

Das Defaultpackage heißt `main`.

Variablen aus einem anderen `package` werden durch `Paketname::$Var` angesprochen.

Geltungsbereich innerhalb einer Funktion:

```
my $Var;
my ($Var1, $Var2, ..)
local ($Var1, $Var2, ..)
```

Nicht mehr gebrauchte Variablen werden in Perl mittels *Garbage-Collection* entfernt!

## Operatoren

Entsprechen weitestgehend denen in der Sprache ‚C':

| | |
|---|---|
| **Nummerisch** | `+ - * / % ++ –` |
| **Vergleich** | `<= == != >=` |
| **Zuweisung** | `= += -= *= /=` |
| **Logisch** | `&& \|\|` |

Zusätzlich gibt es weitere Operatoren:

| | | |
|---|---|---|
| **Nummerisch** | `**` | **Exponentiell: 2**3 ergibt 8** |
| **Vergleich** | `<=>` | **($a <=> 5) gibt -1,0 oder 1 zurück** |
| **String-vergleich** | `lt gt eq le ge ne comp` | **comp entspricht nummerisch** |
| **Logisch** | `and or xor not` | **sind möglich** |
| **String-verkettung** | `$String = $String1 . $String2` | **$String2 wird an $String1 angefügt** |
| **String-vervielfachung** | `$String = $String x 5` | **$String wird verfünffacht** |

## Kontrollstrukturen

```
if (Bed) { Befehle; }
Befehl if (Bed);
unless (Bed) { Befehle; }    # wird ausgeführt, wenn die Bed nicht wahr ist
Befehl unless (Bed);
if (Bed) { Befehle; } else { Befehle; }
if (Bed1) { Befehle; } elseif (Bed2) { Befehle;} else { Befehle; }
while ( LaufBed) { Befehle; }
until (AbbruchBed) { Befehle; }
do { Befehle; } while (LaufBed);
do { Befehle ;} until (AbbruchBed);
for (Initialisierung; Laufbedingung; DurchlaufEnde) { Befehle; }
foreach $LaufVar (@Liste) { Befehle; }
```

### Beispiele

```
print "Wert = 1" if ($wert == 1);
for ($i=2; $i<=10; $i+=2) {print $i, "\n"}
    # geraden Zahlen 2 bis 10 zeilenweise
(foreach $wert (@Tina) { print $wert; }
    # gibt die Werte des Arrays Tina aus
```

**Anmerkung:** Eine Schleife kann mit dem Befehl `last` vorzeitig beendet werden. Mit dem Befehl `next` kann der momentane Schleifendurchgang beendet und der nächste Durchgang gestartet werden. Mit dem Befehl `redo` kann der momentane Schleifendurchgang vorzeitig beendet und nochmals gestartet werden.

## Unterprogramme

### Deklaration

Unterprogramme können überall im Programm erklärt werden (auch mitten in Schleifen).

```
Sub Name
{ Befehle;}
```

### Aufruf

Erfolgt durch:

```
&Name(Werteliste); # '&' kann entfallen, wenn das Unterprogramm vor dem Aufruf erklärt wurde
```

### Wertrückgabe

Unterprogramme geben immer einen Wert zurück! Dieser Wert ist der im Unterprogramm zuletzt evaluierte Wert. Um sicher zu gehen, dass auch der richtige Wert zurückgegeben wird, sollte als letzte Zeile stehen:

```
$Rueckgabewert;
```

### Vorzeitiges Beenden

Die Rückgabe eines Wertes erfolgt mit

```
return ($Rueckgabewert);
```

### Lokale Variablen

Sind nur im Unterprogramm gültig.

```
local ($Var1, @Var2, $Var3);
my ($Var1, @Var2, $Var3);
```

### Parameterübergabe

Beim Aufruf werden die Werte (einfache Variablen oder Arrays) in Klammern mitgegeben. Sie werden im speziellen Array "`@_`" zwischengespeichert und können im Unterprogramm ausgelesen werden. Die Anzahl und der Typ der Werte ist frei und kann von Aufruf zu Aufruf verschieden sein.

### Übergabe per Wert

```
$a=1; $b=2; $c=3;
$Summe = &Summiere($a, $b, $c);   # es werden nur Kopien der Werte übergeben
print ($Summe);      # die Zahl 6 wird ausgegeben

sub Summiere
{  local ($Zahl1, $Zahl2, $Zahl3) = @_; # Ändern von $Zahl1 würde $a unverändert lassen!
    return ($Zahl1 + $Zahl2 + $Zahl3);
}
```

### Übergabe per Referenz

```
$a=1; $b=2;
&Vertausche(*$a, *$b);  # es werden die Adressen übergeben
print ($a, $b);     # die Werte von $a und $b sind vertauscht

sub Vertausche
{  local (*Zahl1, *Zahl2) = @_;   # $Zahl1 ist nun ein anderer Name für $a
    local ($Temp);
   $Temp=$Zahl2; $Zahl2=$Zahl1; $Zahl1=$Temp;
# $a und $b werden geändert
}
```

Unterprogramme können andere Unterprogramme und auch sich selbst aufrufen.

### Spezielle Unterprogramme

```
BEGIN { Befehle; } # wird von Perl immer am Programmanfang gestartet
END { Befehle; }   # wird von Perl immer unmittelbar vor Programmende gestartet
AUTOLOAD { Befehle; }   # wird von Perl immer dann gestartet, wenn ein aufgerufenes anderes Programm nicht vorhanden ist
```

## Funktionsbibliotheken

Sind Sammlungen von Funktionen in einer eigenen Textdatei mit der Namensendung `.pl`. Einzige Voraussetzung ist, dass die letzte Zeile der Bibliotheksdatei einen von 0 verschiedenen Wert ergibt! Es gibt viele vorgefertigte Standardbibliotheken in einer Standardperlinstalation (**`bigint.pl, perldb.pl, ..`**).

Vor dem Aufruf einer solchen Funktion in einem PERLScript, muss man folgende Zeile schreiben:

```
require („Bibliotheksdateiname“); # die ganze Datei wird inkludiert!
```

## Module

Sind ebenfalls Sammlungen von Funktionen in einer eigenen Textdatei mit der Namensendung `.pm`. Zum Unterschied zu einer Funktionsbibliothek, bei der nur die ganze Datei geladen werden kann, kann man aus einem Modul gezielt eine bestimmte einzelne Funktion inkludieren. Die Moduldefinition erfordert ein paar Zeilen Perlcode, die bestimmen, welche Funktionen und welche Variablen exportiert werden dürfen. Es gibt viele vorgefertigte Standardmodule (**`integer.pm, socket.pm, ..`**). Der Aufruf erfolgt mit

```
use Modulname;
```

## Zugriff auf Dateien und Daten

### Textdatei zum Lesen öffnen

```
open(DATEI, "Dateiname") die („Kann Datei nicht öffnen!\n“);
# erzeugt Filehandle "DATEI"
$Zeile = <Datei> # erste Zeile wird gelesen
while ($Zeile ne "")
  # solange die nächste eingelesene Zeile noch Text enthält
{ print ($Zeile);  # gib Zeile aus
}
```

Eine vereinfachte Form gibt es, wenn man die Zeilen einer Datei in ein Array liest:

```
open(DATEI, "Dateiname");
@Liste = <Datei> ; #jede Zeile der DATEI ist nun ein Listenelement
```

### Textdatei zum Schreiben öffnen

```
open(DATEI, ">Pfad/Dateiname");   # Datei wird neu angelegt, alte Daten werden gelöscht
foreach $Eintrag (@Liste)         # $Eintrag durchläuft eine vorgegebene Liste
{ print DATEI ($Eintrag); }  # Schreibt die Variable $Eintrag in die DATEI
```

### Textdatei zum Daten Anfügen öffnen

Erfolgt wie das Beschreiben, jedoch mit ‘`>>`’

```
open(DATEI, ">>Pfad/Dateiname");
```

### Datei schließen

```
close (DATEI};
```

### Standard Input- und Output-Datei

Mit `<STDIN>` und `<STDOUT>` sind im allgemeinen Tastatur und Monitor gemeint. Sie brauchen weder geöffnet noch geschlossen zu werden.

Achtung: Bei CGI-Scripts übernimmt der Server die Rolle aller dieser Dateien.

```
$Zeile =  <STDIN>;       # liest eine Zeile von der Standardeingabe
print STDOUT ($Zeile);  # dasselbe wie print $Zeile;
```

### DATA

Analog zu einem Assemblerprogramm kann man am Ende eines Programmcodes Daten festlegen.

```
$Daten = <DATA>;    # liest die erste Zeile der Daten
print $Daten;       # ergibt „Hier sind die Datenzeilen ..“
__END__         # kennzeichnet das Ende des Programmcodes (muss sein!)
Hier sind die Datenzeilen .. # die Daten kommen hierher
```

### Pipes mit Dateien bilden

```
open (MESSAGE, „| mail donald“);  # ein Pipe zur Applikation mail eröffnen
  print MESSAGE “Hallo Donald! Hier ist das Perl-Demoprogramm zum mail-Versenden!“;
close (MESSAGE);
```

oder:

```
open (DATEN, „ls -la |“);    # die von ls -la erzeugte Ausgabe wird in DATEN umgeleitet
```

### Low Level Dateizugriffe

Wird eine Datei wie oben geöffnet, dann stehen auch folgende Befehle zur Verfügung:

```
read(FILEHANDLE, $Var, Laenge_in_byte)
# gepuffert (wie 'fread' in C)
sysread(FILEHANDLE, $Var, Laenge_in_Byte)
# ungepuffert (wie 'read' in C)
syswrite(FILEHANDLE, $Var, Laenge_in_Byte)
# ungepuffert (wie 'write' in C)
seek(FILEHANDLE, Offset_in_Byte_von_Position, Position)
# wie 'fseek' in C
# dabei bedeutet Position 1=Dateianfang, 2=aktuelle Position, 3=Dateiende
tell FILEHANDLE:    # gibt die aktuelle Dateizeigerposition in Byte
@Info = stat(FILEHANDLE);    # gibt 13 verschiedene Informationen über die Datei, wie InodeNr oder uid
```

### Datei-Testoperatoren

Es gibt mehr als 25 verschiedene Testoperatoren für alle möglichen Daten, die mit Dateien zusammenhängen:

```
if (-e „Dateiname“) {print „Datei existiert!“};
if (-d „Dateiname“) {print „Datei ist ein Verzeichnis!“};
if (-r „Dateiname“) {print „Datei ist von diesem Programm lesbar!“};
if (-w „Dateiname“) {print „Datei ist von diesem Programm beschreibbar!“};
if (-T „Dateiname“) {print „Datei ist ein Textfile!“};
```

## Zugriff auf Verzeichnisse

```
opendir(VERZ, “/path/mydir”); # öffnet Verzeichnis mit Filehandle VERZ
@Dateiliste = readdir(VERZ); # die Dateinamen sind in der @Dateiliste
closedir(VERZ);
```

## Funktionen mit Strings

Hier sind nur einige wenige:

```
$String = chop($String);     # entfernt das letzte Zeichen des Strings
$String = chomp($String);    # entfernt ein anhängendes RETURN, falls es dieses gibt
$num = length($String);
$position = index($String, $Substring);
# ergibt Position des Substring, sonst 0
$Substring = substr($String, AnfangsPos, Laenge);
# liefert einen TeilString
@Zeit = split(/:/, "Stunde:Minute:Sekunde");
# liefert („Stunde“, „Minute“, „Sekunde“)
if ($Frage =~ /bitte/) { print „Danke“;}
# prüft $Frage, ob es den String „bitte“ enthält
$string =~ s/a/A/; # ersetze in $string das erste Zeichen ‘a’ durch ‘A’ (s .. ‘substitute’)
$string =~ s/a/A/g;# ersetze in $string alle Zeichen ‘a’ durch ‘A’ (g .. ‘global’)
$string =~ s/a[0-9]/b/; # ersetze in $string alle Zeichenfolgen ‘a0’,’a1’, .. ‘a9’ durch ‘b’
$string =~ s/a.*b/c/;   # ersetze in $string alle Zeichenfolgen, die mit ‘a’ anfangen, mit beliebig vielen (*) beliebigen (.) Zeichen (ausser ‘\n’)fortsetzen und mit ‘b’ enden mit ‘c’
if ($string =~ /^Bitte/) {print ‘$string fängt mit \"Bitte\" an’;}
if ($string =~ /$bitte/) {print ‘$string hört mit \"bitte\" auf’;}
```

## mehrzeilige Textausgabe

```
print <TEXTENDE;   # schreibt einen mehrzeiligen Text
... auszugebende Textzeilen...
TEXTENDE
```

Die Definition eigener Textausgabeformate ist möglich!

Analog zu C gibt auch den printf()-Befehl.

```
printf Filehandle (Formatstring, Werteliste);
# wie in C
```

## Prozesse starten und beenden

### eval(string)

```
$String = „print ‚Hallo!'“;  eval($String);
# wandelt $String in eine Perl-Anweisung um und führt diese aus
```

### system(@Befehlsliste)

```
@Befehl = („ls“, „-la „, „Verzeichnisname“);
system(@Befehl);    # führt den in @Befehl angegebenen Systembefehl
mit seinen Optionen und Parametern aus. Das laufende Perlprogramm wird
bis zum Ende des aufgerufenen Programms unterbrochen und anschließend
weitergeführt
```

### exec(@Befehlsliste)

wie `system(@Befehlsliste)`, jedoch wird das Perlprogramm vor dem Aufruf des anderen Programms beendet, die weiteren Perlbefehle werden nicht mehr ausgeführt.

### $ProzessNr = fork(@Befehlsliste)

wie `system(@Befehlsliste)`, jedoch läuft sowohl die Abarbeitung des Perlprogramms als auch die des aufrufenden Programms parallel weiter.

### die($Message)

beendet das laufende Perlprogramm mit der angegebenen Meldung.

### exit(Exitcode)

beendet das laufende Perlprogramm mit dem angegebenen Exitcode.

### kill(Signal, Prozessliste)

sendet das Signal an alle Prozesse der Liste. Zum Beispiel

```
kill(9,12)     # beendet den Prozess mit PID 12
```

## Kommandozeilen-Optionen

Mit den vielen Optionen von Perl kann man das Verhalten des Programmablaufs modifizieren. Zum Beispiel:

### Zeilenweises Abarbeiten mehrerer Textdateien

Das folgende Programm mit dem Namen „**ausgabe.pl**“ wird für jede Zeile der Textdateien ausgeführt.

Aufruf: **ausgabe.pl Textdatei1 Textdatei2 ..**

```
#!/usr/bin/perl -n
$Zeile = $ ;   # nächste Zeile einlesen
print $Zeile
```

### Ausführen eines Perlprogramms mitten aus einer Textdatei

Dies erlaubt das Testen eines Programms aus zum Beispiel einer Hilfedatei.

Aufruf: **perl –x Textdatei**

Inhalt der Textdatei:

```
Hier ist die Textdatei aus deren Mitte ein Perlcode
ausgeführt werden soll. Der Text hier ist reiner Platzfüller.
Beginnen wir mit dem Programmcode ...
#!/usr/bin/perl
  print „Hallo!!\n";   # hier steht der Programmcode
__END__
Viel Spass beim Ausführen!
```

## PERL Debugger

Zur Fehlersuche in PERLScripts. Der Aufruf erfolgt mit

**perl –d Scriptname**

Die einzelnen Befehle werden in der Kommandozeile in Kurzform eingegeben. Jede Eingabe, die nicht als spezielle Debuggeranweisung interpretiert werden kann, wird als PERL-Befehl ausgeführt!

| | |
|---|---|
| **l oder l 10-17** | **listet die nächsten Scriptzeilen auf** |
| **L** | **listet die nächste auszuführende Scriptzeile auf** |
| **s** | **führt die nächste Befehlszeile aus** |
| **n** | **wie s, jedoch mit Sprung in eine evtl. Subroutine** |
| **RETURN** | **wiederholt die letzte Eingabe** |
| **r** | **Beenden des momentanen Subroutine** |
| **x** | **listet die aktuellen Variableninhalte aus** |
| **b ZeilenNr** | **setzt Breakpoint** |
| **c** | **Programmlauf bis zum nächsten Breakpoint oder zum Ende** |
| **d ZeilenNr** | **löscht Breakpoint** |
| **t** | **trace on und off** |
| **H** | **History der Debuggerkommandos** |
| **h** | **help** |
| **q** | **quit** |

## Literatur


- Programming PERL, Larry Wall and R.L.Schwartz, O'Reilly & Associates (Die PERL-Bibel)
- Learning PERL, R.L.Schwartz and Tom Christansen, O'Reilly & Associates
- Teach Yourself PERL 5 in 21 Days, David Till, SAMS Publishing
- PERL by Example, Ellie Quigley, Prentice Hall

und viele andere …


## Online Kurse

- RRZK Perlkurs, Farid Hajji, ) (in Deutsch auch zum Download)
  ***http://gd.tuwien.ac.at/languages/perl/Hajji-Perlkurs/***
- Links zu PERL TUTORIALS,
  ***http://www.perl.com/reference/query.cgi?tutorials***

## Tools in PERL

- CPAN, ***http://gd.tuwien.ac.at/languages/perl/CPAN/***
- Alles was man sich wünscht, ***http://www.perl.com/reference/***

*Es gibt Leute, die halten einen Unternehmer für einen räudigen Wolf, den man totschlagen müsse. Andere wieder meinen, er wäre eine Kuh, die man ununterbrochen melken könne. Nur wenige sehen in ihm das Pferd, das den Karren zieht.*

*Sir Winston Churchil*