Einführung in das Programmieren mit Perl

Bernd Ulmann[a]

01-SEP-2004

Jede kommerzielle Nutzung der folgenden Folien ist untersagt.

Die Veröffentlichung, auch auszugsweise, bedarf der Zustimmung des Autors.

[a] ulmann@vaxman.de

Allgemeines

- Perl wurde von Larry Wall entwickelt.
- Perl ist *kein* Akronym, aber ein Retronym:
  1. *P*ractical *E*xtraction *R*eport *L*anguage
  2. *P*athologically *E*clectic *R*ubbish *L*ister
- Perl ist eine ausgesprochen mächtige und leistungsfähige Interpretersprache
- Alles rund um Perl findet sich auf *CPAN*, dem *C*omprehensive *P*erl *A*rchive *N*etwork:

  `http://www.cpan.org`

- Die Grundphilosophie lautet:

  There is more than one way to do it.

Ausführen von Perl-Programmen

- Perl-Programme werden in der Regel in Textfiles abgelegt.
- Die Filenamenendung von Perl-Programmen sollte `.pl` lauten.
- Die einfachste Art, ein Perl-Programm auszuführen, besteht darin, den Perl-Interpreter mit dem Filenamen des auszuführenden Programmes von der Kommandozeile aus zu starten:

```
perl hello_world.pl
```

Hello World

❶ `print 'Hello world!';`

- Kein Zeilenumbruch nach erfolgter Ausgabe.

❷ `print "Hello world!\n";`

- `\n` ist ein Steuerzeichen
- Doppelte Anführungszeichen werten – im Gegensatz zu einfachen – unter anderem solche Steuerzeichen aus.

Hello World

❸
```
$hw = 'Hello world!';
print "$hw\n";
```

- $hw ist eine skalare Variable – in diesem Fall also eine Zeichenkette, ein *String*.
- Doppelte Anführungszeichen werten Variablen, die sich zwischen ihnen befinden, aus!
- Was hier fehlt, ist eine Variablendeklaration – so, wie hier gezeigt, wird die Variable durch ihre bloße Verwendung eingeführt, was sehr fehlerträchtig ist!

Hello World

❹
```
use strict;
my $hw = 'Hello world!';
my $empty;
print "$hw\n$empty\n";
```

- Durch `use strict;` wird erzwungen, daß Variablen *vor* ihrer Verwendung durch eine Deklaration mit `my` vereinbart werden.
- Ein Programm sollte darum *immer* als erste Anweisung ein `use strict;` enthalten.
- Leider kann immer noch ohne Fehlermeldung und Warnung auf zwar deklarierte, aber nicht mit einem Wert belegt Variablen, wie hier `$empty` zugegriffen werden. `use strict;` alleine genügt also nicht.

Hello World

```
❺ use strict;
  use warnings;
  my $hw = 'Hello world!';
  my $empty;
  print "$hw\n$empty\n";
```

- `use warnings;` generiert Warnmeldungen des Interpreters bei dem Versuch, auf nicht initialisierte Variablen lesend zuzugreifen:

```
Use of uninitialized value in concatenation (.)
or string at hello_world.pl at line 5.
Hello world!
```

Fazit

- Die Filenamen von Perl-Programmen sollten stets auf `.pl` enden.
- Jedes Perl-Programm sollte mit

```
use strict;
use warnings;
```

  beginnen!

- Doppelte Anführungsstriche *nur* dann verwenden, wenn Steuerzeichenketten oder Variablen innerhalb dieser ausgewertet werden sollen. Anderenfalls nur einfache Hochkommata verwenden (*Quotes*).

Variablen in Perl

- Perl unterscheidet – im Gegensatz zu den meisten anderen Programmiersprachen – *nicht* zwischen verschiedenen Variablentypen wie `int`, `float`, etc.
- Perl kennt jedoch drei verschiedene Strukturen für Variablen:
  1. *Skalare*
  2. *Arrays* (auch *Felder* genannt)
  3. *Hashes* (auch als *assoziative Felder* bezeichnet)

Skalare

- Skalare Variablen beginnen stets mit einem $.
- Skalare Variablen sind dadurch gekennzeichnet, daß eine Variable einen einzelnen Wert enthält.
- Dieser Wert selbst kann jedoch beliebig sein:

```
use strict;
use warnings;
my $pi = 3.1415926535;
my $hw = 'Hello world!';
my $answer = 42;
```

- Ein Skalar kann also sowohl eine Gleitkommazahl, als auch eine ganze Zahl, als auch eine Zeichenkette (einen String) enthalten.

Arrays

- Arrayvariablen beginnen stets mit einem @.
- Ein Array ist ein Feld, dessen Elemente Skalare sind:

```
use strict;
use warnings;

my @feld;
$feld[0] = ’Erstes Element’;
$feld[1] = ’Zweites Element’;

print "@feld\n";
print "’$feld[0]’ und ’$feld[1]’\n";
```

Arrays

- Die Numerierung von Arrayelementen beginnt bei 0.
- Was gibt folgendes Programm aus?

```
use strict;
use warnings;

my @feld;
my $feld[0] = 'Erstes Element';
my $feld[2] = 'Zweites Element';

print "@feld\n";
```

Da `$feld[1]` nicht belegt wurde, jedoch durch das `print`-Statement ausgegeben wird, wird eine Warning generiert.

Arrays

- Die Operatoren `push`, `pop`, `shift` und `unshift`:

```
use strict;
use warnings;
push my @feld, ’gepushtes Element’;
unshift @feld, ’geunshiftetes Element’;
print shift @feld, "\n", pop @feld, "\n";
```

- Die Ausgabe lautet:

```
geunshiftetes Element
gepushtes Element
```

- Was tun die vier genannten Operatoren?

Arrays

- Arrays können auch einfacher mit Werten belegt werden:

```
use strict;
use warnings;

my @feld_1 = 5..14;
my @feld=2 = qw{eins zwei drei};

print "@feld_1\n@feld_2\n";
```

Fazit

- `push` fügt neue Elemente von rechts an ein Array an,
- `pop` entfernt Elemente eines Arrays von rechts.
- `unshift` und `shift` arbeiten analog, nur auf der linken Seite des Arrays.
- `qw`, kurz für *Quoted Words*, erzeugt eine Liste aus Wörtern, die beispielsweise direkt einem Arrays zugewiesen werden kann.
- `start..end` erzeugt eine arithmetische Progression, beginnend mit dem Wert `start` und endend mit dem Wert `end`.
- Übrigens: Variablen können an fast jeder Stelle mit `my` vereinbart werden.

Hashes

- Hashvariablen beginnen stets mit einem %.
- Hashelemente werden über *Schlüssel* angesprochen:

```
use strict;
use warnings;
my %h;
$h{’Vorname’} = ’Bernd’;
$h{’Nachname’} = ’Ulmann’;
print "%h\n";
```

- Die einfache Ausgabe mit print funktioniert hier jedoch *nicht*, da die Hashelemente quasi nicht wie in einem Array „hintereinander“ liegen.

Hashes

- Hashes können auch einfacher mit Werten belegt werden:

  ❶ Zuweisung im Listenkontext:

```
my %h = ('Vorname', 'Bernd', 'Nachname', 'Ulmann');
```

  ❷ Arrow-Zuweisung:

```
my %h = (
         'vorname' => 'Bernd',
         'nachname' => 'Ulmann',
        )
```

Tricks mit Hashes

- Wieviele Elemente hat ein Hash?

```
my %h = ('Vorname', 'Bernd', 'Nachname', 'Ulmann');
my $anzahl = keys %h;
```

Die Verwendung einer Liste im skalaren Kontext liefert die Anzahl ihrer Elemente zurück.

- Invertieren eines Hashes: Manchmal wäre es praktisch, die Zuordnung zwischen Schlüsseln und Werten in einem Hash zu invertieren, um beispielsweise eine Telefonnummernrückwärtssuche zu ermöglichen:

```
my %tel = ('Ulmann', '314159', 'Meier', '271828');
my %inv = reverse %tel;
```

Erklärung

- Die Funktion `reverse` kehrt die Reihenfolge der Elemente einer Liste um – die Verwendung von `%tel` im Listenkontext dieser Funktion ergibt zunächst die Liste

  ```
  'Ulmann', '314159', 'Meier', '271828'
  ```

- `reverse %tel` liefert nun entsprechend

  ```
  '271828', 'Meier', '314159', 'Ulmann'
  ```

- Die Zuweisung dieser Liste zu einem Hash, in diesem Fall `%inv` erzeugt nun die gewünschte Datenstruktur, die Telefonnummern Namen zuordnet.
- *Vorsicht:* Wenn eine Telefonnummer mehreren Namen zugeordnet ist, überschreiben sich diese, so daß nur der letzte erhalten bleibt!

Hashes

- Das folgende Programm belegt zwei Hashelemente und gibt im Anschluß hieran alle Elemente, sortiert nach ihren Schlüsseleinträgen wieder aus:

```
use strict;
use warnings;

my %h;
$h{'Vorname'} = 'Bernd';
$h{'Nachname'} = 'Ulmann';
for my $key (sort (keys (%h)))
{
  print "Schluessel: $key ---> $h{$key}\n";
}
```

Die verwendete `for`-Schleife

- Die Funktion `keys ()` liefert die Liste aller Schlüssel eines Hashes zurück.
- Die Funktion `sort ()` liefert die Elemente der als Parameter übergebenen Liste in alphabetisch sortierter Reihenfolge zurück.
- `(sort (keys (%h))` liefert also eine sortierte Liste aller Schlüssel des Hashes `%h` zurück.
- Die Schleife `for my $key (sort (keys (%h)))` durchläuft alle Elemente der Schlüsselliste. Für jedes Element werden die zwischen den direkt folgenden geschweiften Klammern stehenden Instruktionen ausgeführt.
- *Bemerkung*: Analog zur Funktion `keys` existiert auch eine Funktion `values`, die eine Liste aus allen in einem Hash enthaltenen Werten zurückliefert.

Hashes

- Eine andere Möglichkeit, alle Einträge eines Hashes zu durchlaufen und gegebenenfalls auszugeben, ist folgende:

```
use strict;
use warnings;

my %h;
$h{'Vorname'} = 'Bernd';
$h{'Nachname'} = 'Ulmann';
while (my ($key, $wert) = each %h)
{
  print "Schluessel: $key ---> $h{$key}\n";
}
```

Erklärung

- Die Funktion `each` liefert, angewandt auf einen Hash, bei jedem Aufruf ein neues Paar aus Schlüssel und zugeordnetem Wert zurück.
- Die `while`-Schleife wird so lange durchlaufen, wie `each` Elemente zurückliefert.

Blöcke, Schleifen und andere Kontrollstrukturen

- Logisch zusammenhängende Instruktionen, die beispielsweise in einer Schleife oder unter einer bestimmten Bedingung ausgeführt werden sollen, werden durch geschweifte Klammern zu sogenannten *Blöcken* zusammengefaßt.
- Beispiel: Berechne die Summe von 1 bis 100:

```
use strict;
use warnings;
my $summe = 0;
for my $wert (1..100)
{
  $summe = $summe + $wert;
}
print "Summe = $summe\n";
```

Der Scope von Variablen

- Eine Variable ist nur innerhalb des Blockes, innerhalb dessen sie deklariert wird, sichtbar:

```
use strict;
use warnings;

my $aussen = 0;
for my $i (1..2)
{
  my $innen = 1;
}

print "$aussen, $i, $innen\n";
```

Der Scope von Variablen

- Die Variable `$aussen` ist innerhalb des gesamten Programmes sichtbar, da sie im äußersten Block deklariert wurde.
- Die Variablen `$i` und `$innen` sind nur innerhalb der `for`-Schleife sichtbar, nicht jedoch außerhalb.
- Der Versuch, das Programm auszuführen, scheitert mit folgenden Fehlermeldungen:

```
Global symbol "$i" requires explicit package
name at scope.pl line 10. Global symbol "$innen"
requires explicit package name at scope.pl line
10. Execution of scope.pl aborted due to
compilation errors.
```

Mehr Schleifen und Kontrollstrukturen

❶ `for`-Schleifen:

- Zur Verarbeitung von Listen
  (`$summe += $i;` entspricht `$summe = $summe + $i;`):

```
my $summe = 0;
for my $i (1..10){
  $summe += $i;
}
```

- C-artige Variante (`$i++` entspricht `$i = $i+1`):

```
my $summe = 0;
for (my $i = 1; $i < 11; $i++){
  $summe += $i;
}
```

Mehr Schleifen und Kontrollstrukturen

❷ `while`-Schleifen:

```
my $summe = 0;
my $i = 1;
while ($i < 11)
{
  $summe += $i;
  $i++;
}
```

Mehr Schleifen und Kontrollstrukturen

❸ `until`-Schleifen:

```
my $summe = 0;
my $i = 1;
until ($i > 10)
{
  $summe += $i;
  $i++;
}
```

Mehr Schleifen und Kontrollstrukturen

❹ `if-else`-Strukturen, etc.:

- Einfaches `if-else`:

```
my $i = 5;
if ($i < 5)
{
  print "i ist kleiner als 5.\n";
}
else
{
  print "i ist groesser gleich 5.\n";
}
```

Mehr Schleifen und Kontrollstrukturen

❹ `if-else`-Strukturen, etc.:

- `if` mit negierter Bedingung:

```
my $i = 5;
if (!($i < 5))
{
}
```

- `unless`:

```
my $i = 5;
unless ($i < 5)
{
}
```

Mehr Schleifen und Kontrollstrukturen

❹ `if-else`-Strukturen, etc.:

- `if` bzw. `unless` mit nur einer einzigen kontrollierten Operation:

  Statt

```
my $i = 5;
if ($i < 5)
{
  print "i ist kleiner 5.\n";
}
```

  ist

```
print "i ist kleiner 5.\n" if $i < 5;
```

  kürzer und besser lesbar.

Mehr Schleifen und Kontrollstrukturen

❹ `if-else`-Strukturen, etc.:

- Die `elsif`-Konstruktion:

```
my $i = 5;
if ($i < 5) {
  print "i ist kleiner 5.\n";
}
elsif ($i < 10) {
  print "i ist groesser 4 und kleiner 10.\n";
}
else {
  print "i ist groesser gleich 10.\n";
}
```

Mehr Schleifen und Kontrollstrukturen

❺ Zusätzliche Steuerungsmechanismen für Schleifen:

- Soll eine Schleife bei Eintreten einer bestimmten Bedingung vorzeitig verlassen werden, kann dies durch die `last`-Operation geschehen:

```
use strict;
use warnings;

for my $i (1..10)
{
  last if $i > 5;
  print "$i\n";
}
```

Mehr Schleifen und Kontrollstrukturen

- Zusätzliche Steuerungsmechanismen für Schleifen:
  - Soll ein Schleifendurchlauf bei Eintreten einer bestimmten Bedingung beendet und mit dem nächsten fortgefahren werden, kann dies durch die `next`-Operation geschehen:

```
use strict;
use warnings;

for my $i (1..10)
{
  next if $i == 5;
  print "$i\n";
}
```

Fazit

- Perl verfügt über eine Reihe verschiedener Schleifen- und Bedingungskonstruktionen.
- Variablen sind stets nur innerhalb des Blockes, in welchem sie deklariert werden, sichtbar. Aus Wartbarkeitsgründen, sollte hiervon reger Gebrauch gemacht werden – globale Variablen, d.h. Variablen, die außerhalb aller anderen Blöcke deklariert werden, sind nur in begründeten Fällen zu verwenden!
- Soll eine Bedingung nur auf eine Operation wirken, kann auf die geschweiften Blockklammern verzichtet werden, indem die Bedingung hinter die betreffende Operation geschrieben wird.
- `unless (<bedingung>)` verhält sich wie `if (!<bedingung>)`, wobei das Ausrufungszeichen die Negationsoperation darstellt.

Ein-/Ausgabe von Daten

- Ausgabe von Daten:
  - In der Regel werden Daten mit Hilfe einer `print`-Anweisung ausgegeben.
    Eine solche Anweisung kann sich über mehrere Zeilen erstrecken:

```
print "Dies ist ein mehrzeiliges
print-Statement.
";
```

  - Ohne zusätzliche Angaben werden Ausgaben von `print` auf die sogenannte *Standardausgabe*, kurz *stdout*, geschrieben.

Ein-/Ausgabe von Daten

- Einlesen von Daten:
  - Um Daten von der sogenannten *Standardeingabe*, kurz *stdin*, zu lesen, kann wie folgt vorgegangen werden:

```
use strict;
use warnings;

print ’Bitte geben Sie Ihren Namen ein: ’;
my $name = <STDIN>;
chomp $name;
print "Hallo, $name.\n";
```

Ein-/Ausgabe von Daten

- Einlesen von Daten:
  - Die Angabe `<STDIN>` spezifiziert die Standardeingabe als Datenquelle. `$name = <STDIN>;` liest so lange Zeichen von der Standardeingabe, bis die *Return*-Taste betätigt wird.
  - Diese Eingabe wird als String in `$name` abgelegt – zusammen mit dem abschließenden Zeilenumbruch, der durch Betätigen der *Return*-Taste erzeugt wurde.
  - Die Funktion `chomp` entfernt einen Zeilenumbruch am Ende eines Strings, falls ein solcher Umbruch vorhanden ist. Nach `chomp $name;` enthält `$name` nur noch die eingegebenen Nutzzeichen, jedoch nicht mehr den abschließenden Zeilenumbruch.

Beispiel – Summenberechnung

- Berechne die Summe von Zahlen, ihrer Quadrate und Kuben:

```
use strict;
use warnings;

my ($summe, $summe_quadrat, $summe_kubus) = (0, 0, 0);

print 'Berechne Summen bis: ';
my $ende = <STDIN>;
chomp $ende;

for my $i (1..$ende)
{
  my $quadrat = $i * $i;
  my $kubus = $quadrat * $i;

  $summe += $i;
  $summe_quadrat += $quadrat;
  $summe_kubus += $kubus;

  print "$i\t$quadrat\t$kubus\n";
}

print "-------------------
$summe\t$summe_quadrat\t$summe_kubus\n";
```

Bemerkungen und Probleme

- Listen können in Perl einander direkt zugewiesen werden – im vorliegenden Beispiel werden in einer Zeile drei Variablen deklariert und jeweils mit dem Initialwert `0` belegt.
- Dies kann unter anderem sehr effizient zum Vertauschen der Inhalte zweier Variablen ohne dritte Hilfsvariable Verwendung finden:

```
($a, $b) = ($b, $a);
```

- Die Steuerzeichenkette `\t` gibt ein *TAB* aus, springt also zur nächsten Tabulatormarke.
- Problematisch ist die Benutzereingabe – wird ein negativer Wert oder gar etwas, das keine ganze Zahl darstellt, eingegeben, kann das Programm abstürzen. Eine verbesserte Programmversion sollte also sicherstellen, daß Fehleingaben keine unerwarteten Programmreaktionen zur Folge haben.

Einfache reguläre Ausdrücke

- In vielen Fällen ist es nötig, eine Zeichenkette auf die Einhaltung bestimmter Forderungen hinsichtlich ihrer Gestalt zu untersuchen. Im vorliegenden Programm muß für eine gültige Benutzereingabe gelten, daß sie ausschließlich aus Ziffern besteht.
- In den meisten anderen Programmiersprachen wäre es nun notwendig, mit einer Schleife jedes einzelne Zeichen der Benutzereingabe daraufhin zu untersuchen, ob es sich um eine Ziffer handelt. Falls eine Nichtziffer gefunden wird, könnte die Schleife vorzeitig beendet werden.
- In der Mehrzahl der Fälle sind die zu überprüfenden Bedingungen, die an Datenformate gestellt werden, jedoch wesentlich komplizierter als im vorliegenden Beispiel, was eine zeichenweise Verarbeitung mit Schleifen und Flagvariablen schnell unpraktikabel werden läßt.

Einfache reguläre Ausdrücke

- Die Stärke von Perl liegt in erster Linie in der Verarbeitung von Zeichenketten – besonders in der Möglichkeit, sogenannte *reguläre Ausdrücke* zu verwenden, welche Regeln für den „korrekten“ Aufbau von Zeichenketten darstellen.
- Reguläre Ausdrücke, meist *regex* genannt, die Kurzform von *regular expression*, stellen eine leistungsfähige Möglichkeit zur Verfügung, die Struktur von Zeichenketten zu beschreiben.
- Im vorliegenden Beispiel soll für eine zulässige Benutzereingabe gelten, daß sie ausschließlich aus Ziffern besteht, von denen die erste nicht 0 sein darf.

Einfache reguläre Ausdrücke

- Ein regulärer Ausdruck, der solche Zeichenketten beschreibt, könnte folgende Form besitzen: `/^[1-9]\d*$/`
- Ein regulärer Ausdruck wird meist zwischen zwei Slashes `/` eingeschlossen.
- Die Konstruktion `[1-9]` repräsentiert ein einzelnes Zeichen, das eine Ziffer zwischen 1 und 9 (einschließlich) darstellen darf.
- `\d` repräsentiert eine beliebige Ziffer, also eine Ziffer zwischen `0` und `9`.
- Das Zeichen `*` in `\d*` besagt, daß keine oder eine beliebige Anzahl von Ziffern folgen kann.
- Die Zeichen `^` und `$` kennzeichnen den Beginn bzw. das Ende der beschriebenen Zeichenkette, d.h. im vorliegenden Fall darf vor der ersten Ziffer, die nicht `0` sein darf, kein Zeichen auftreten und nach der letzten Ziffer, abgedeckt durch `\d*`, darf kein weiteres Zeichen folgen.

Eine verbesserte Eingaberoutine

- Eine verbesserte Eingaberoutine für das Summationsprogramm muß nun die Eingabe des Benutzers dahingehend überprüfen, ob sie der durch den eben dargestellten regulären Ausdruck festgelegten Form genügt.
- Ist dies nicht der Fall, ist eine neue Eingabe anzufordern:

```
my $ende = '';
until ($ende =~ /^[1-9]\d*$/)
{
  print 'Berechne Summen bis: ';
  $ende = <STDIN>;
  chomp $ende;
}
```

Erklärung

- Die `until`-Schleife wird so lange ausgeführt, bis die in `$ende` enthaltene Zeichenkette dem regulären Ausdruck `/^[1-9]\d*$/` entspricht.
- Der Vergleichsoperator =~ ist dann erfüllt, wenn die links von ihm in einer Variable stehende Zeichenkette die durch den rechts von ihm stehenden regulären Ausdruck beschriebene Gestalt besitzt.
- Anstelle einer `until`-Konstruktion mit =~ hätte auch ein
  `while ($ende !~ /^[1-9]\d*$/)`
  zum Einsatz kommen können.
- Der Benutzer wird also so lange zur Eingabe eines Wertes aufgefordert, bis dieser ausschließlich aus Ziffern, von denen die erste nicht `0` sein darf, besteht.

Summenberechnung – verbesserte Version

```
use strict;
use warnings;

my ($summe, $summe_quadrat, $summe_kubus) = (0, 0, 0);
my $ende = '';

until ($ende =~ /^[1-9]\d*$/)
{
  print 'Berechne Summen bis: ';
  $ende = <STDIN>;
  chomp $ende;
}

for my $i (1..$ende)
{
  my $quadrat = $i * $i;
  my $kubus   = $quadrat * $i;

  $summe += $i;
  $summe_quadrat += $quadrat;
  $summe_kubus += $kubus;

  print "$i\t$quadrat\t$kubus\n";
}

print "----------------------\n$summe\t$summe_quadrat\t$summe_kubus\n";
```

Einlesen einer VISA-Kartennummer

- Das folgende Programmfragment liest eine VISA-Kartennummer ein, die aus vier jeweils vierziffrigen Ziffernblöcken bestehen muß, die durch Bindestriche getrennt sind:

```
use strict;
use warnings;
my $visa_number = '';
until ($visa_number =~ /^(\d{4}-){3}\d{4}$/)
{
  print 'Bitte Kartennummer eingeben: ';
  $visa_number = <STDIN>;
  chomp $visa_number;
}
print "Die Nummer lautet: $visa_number\n";
```

Erklärung

- Der reguläre Ausdruck

  ```
  /^(\d{4}-){3}\d{4}$/
  ```

  beschreibt also eine Zeichenkette, die aus drei Gruppen, `(){3}`, bestehend aus jeweils vier Ziffern, `\d{4}`, sowie einem Bindestrich, `-`, besteht, die von einer einzigen Gruppe, bestehend aus vier Ziffern, `\d{4}`, gefolgt wird.

- Die Angabe einer Zahl in geschweiften Klammern hinter einem Teilausdruck gibt an, wie oft dieser Ausdruck in der beschriebenen Zeichenkette hintereinander auftreten muß.

- Mit Hilfe runder Klammern können (unter anderem) Gruppen von Teilausdrücken zusammengefaßt werden, wie obiges Beispiel zeigt.