



Perl-Programmierer sehen Kamele jeder Art als Maskottchen.

Die London Perl Mongers haben sogar eines aus dem Londoner Zoo adoptiert.

Quelle: He-ba-mue

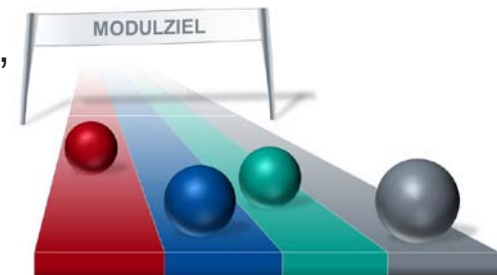
Skriptsprachenorientierte Programmietechnik

Programmiersprache Perl

Prof. Ilse Hartmann

Die Studierenden können nach erfolgreichem Abschluss des Moduls:

- die Eigenschaften der Programmierung von Sprachen der vierten Generation erklären,
- diese, gefestigt durch die praktischen Übungen, in Form der Lösung von programmietechnischen Problemstellungen anwenden,
- erkennen, dass die Sprachen (beispielsweise Perl, Python, Ruby usw.) typischerweise interpreterbasiert sind und eine oft ausrichtungstypische bemerkenswerte Sprachmächtigkeit auf weisen,
- skriptsprachenorientierte Programmierertechnik von den elementaren prozeduralen oder objektorientierten Programmiersystemen unterscheiden,
- die Vorteile scriptorientierter Sprachkonzepte herausstellen z.B. oft flexible Sprachkonzepte bzw. in Form von Bibliotheken in der Regel mögliche Konstrukte auch für komplexe Probleme,
- die Potenz der scriptorientierten Systeme erklären (hohe Entwicklungsperformanz und Plattformunabhängigkeit) und die in der Regel untergeordnete Bedeutung der Ausführungsperformanz nachvollziehen,
- anhand von elementaren und universellen Beispielen zeigen, wie die scriptorientierte Programmierung für Aufgaben aus dem administrativen Bereich bis hin zu hochkomplexen Applikationen sinnvoll eingesetzt werden kann.



Die Modulnote setzt sich folgendermaßen zusammen:

100% Klausur

- Programmieraufgaben
- Hilfsmittel: 4 Seiten selbsterstellte Unterlagen
- Klausurdauer: 120 Minuten
- Bestehensgrenze: muss mindestens mit ausreichend bestanden werden



Anmerkung: Folien beruhen zum Teil auf der Vorlesung von Prof. Havel (FOM München).

- **Schwartz, Randal L., Tom Phoenix: Learning Perl,**
O'Reilly Associates, 3. , Juli 2001. ISBN: 0596001320.
- **Wall, Larry, Tom Christiansen, Jon Orwant: Programming Perl,**
O'Reilly, 2000.
- **Christiansen, Tom, Nathan Torkington: Perl Kochbuch,
Beispiele und Lösungen für Perl-Programmierer.**
O'Reilly Associates Inc., 1999.
- **Ziegler J.: Programmieren lernen mit Perl**
Springer 2002
- **Udo Müller**
Perl Grundlagen, fortgeschrittene Techniken, Übungen
mitp 2007
- **Jürgen Plate**
Der Perl Programmierer
Hanser 2010

1 Einführung in Perl

1.1 1. Perl-Programm

1.2 Daten

1.3 Literale

1.4 Operatoren

1.5 Skalare Variable

1.6 Zuweisungen

1.7 Kontrollstrukturen

1.7.1 Vergleichsoperatoren

1.7.2 Verzweigung

1.7.3 While, do...while -Schleife

1.7.4 For-Schleife

1.8 Warnungen

1.9 Boolesche Werte

- **„Perl“ ist kein Akronym, sondern ein Retronym!**

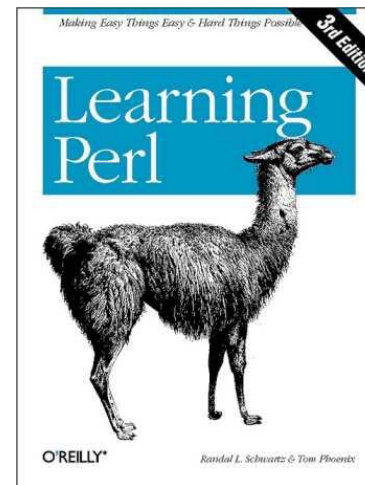
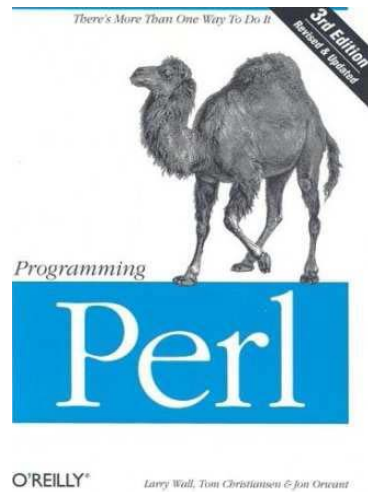
Practical Extraction and Report Language

Pathologically Eclectic Rubbish Lister

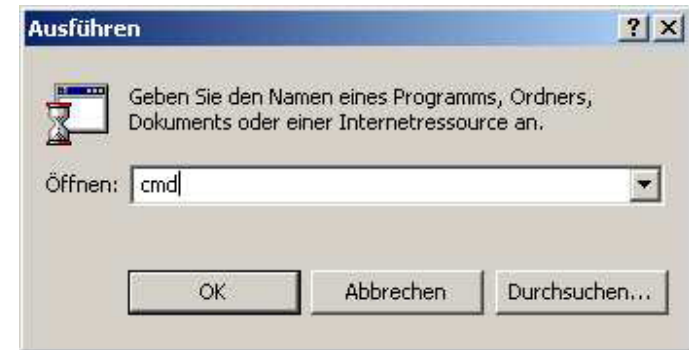
- **Frei verfügbar, plattformunabhängig**
- **Aktuelle Version (Stand 09/2010):**
5.12.1 (Version 6 in Entwicklung)
- **www.perl.org: The Perl Directory (Offizielle Website)**
- **www.cpan.org: Comprehensive Perl Archive Network**



Larry Wall 1987



1. Kommandozeile aufrufen



2. Interpreter starten perl



3. Befehle eingeben print "Hallo Welt!\n";

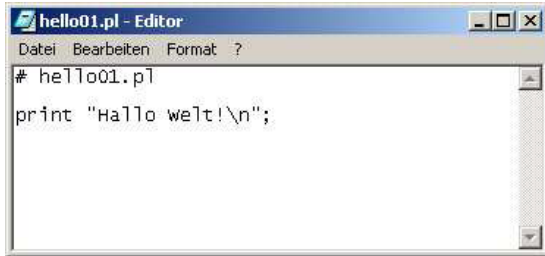


4. Eingabestream beenden

(Strg+Z unter Windows,
Strg+D unter Unix/Linux)

- **Windows Editor (notepad.exe)**
- **UltraEdit (\$50) www.ultraedit.com**
- **PSPad (Freeware) www.pspad.com**
- **Eclipse + EPIC (freie IDE mit Debugging-Tools)
www.eclipse.org / e-p-i-c.sourceforge.net**

- **Quellcode kann mit jedem beliebigem Texteditor geschrieben werden**
- **Kein Textverarbeitungsprogramm (z.B. Word) nutzen!**
- **Speichern als ASCII/ANSI**
- **Dateiendung: Frei wählbar, z.B. keine oder .pl (möglichst nicht .txt!)**
- **Ausführen: Aufruf des Perl-Interpreters (perl) mit Angabe der Quelldatei**



```
# hello01.pl
print "Hallo welt!\n";
```

- Kommentarzeilen werden mit # eingeleitet
- Anweisungen werden (wie in C) per Semikolon begrenzt
- Es wird zwischen Groß- und Kleinschreibung unterschieden (z.B. Fehler bei „PRINT“)
- Leerzeichen, Tabs und Umbrüche sind beliebig einsetzbar
- \n erzeugt einen Zeilenumbruch (d.h. CR+LF unter Windows, auf anderen Systemen teils nur CR oder nur LF)
- Allgemein leitet ein Backslash sog. Escape-Sequenzen ein
- Escape-Sequenzen werden nur in doppelten Anführungszeichen interpretiert („double-quoted strings“)
- Text in einfachen Anführungszeichen wird eins zu eins wiedergegeben („single-quoted strings“)

Perl unterscheidet keine Daten-Typen, sondern nur Strukturen:

- Skalare

- + Einzelne Werte, z.B. Zahlen oder Zeichenketten
- + Zahlen und Strings sind fast beliebig austauschbar
- + Keine Unterscheidung zwischen Ganz- und Kommazahlen
- + Zahlen werden intern stets als Fließkommazahl mit doppelter Genauigkeit (quasi double) gespeichert

- Listen und Arrays

- + Geordnete Ansammlungen von skalaren Werten

- Hashes

- + Sammlung von Werten, die durch Schlüssel indiziert werden

Ein Literal ist die zeichengetreue Darstellung von Daten im Quellcode

- **Ganzzahlige Literale:**

- + 0
- + 2001
- + -40
- + 6512983509457623
- + 6_512_983_509_457_623

- **Fließkomma-Literale:**

- + 1.25
- + 255.0
- + 255.000
- + 7.25e45
- + -6.5E-23

- **Literale anderer Zahlensysteme:**

- + 0377 # 377 oktal = 255 dezimal
- + 0xff # FF hexadezimal = 255 dezimal
- + 0b11111111 # binär, ebenfalls 255 dezimal
- + 0x1377_0B77
- + 0x13_77_0B_77

Strings in einfachen Anführungszeichen

- Jedes Zeichen steht genau für sich selbst
- Ausnahmen: Einfaches Anführungszeichen und Backslash
- Codierung („quoten“) eines einfachen Anführungszeichens: `'`
- Codierung eines Backslashes: `\\`

<code>'Hallo'</code>	# die fünf Zeichen H, a, l, l und o
<code>"</code>	# Nullstring/Leerstring (keine Zeichen)
<code>'Das \'-Zeichen muss gequotet werden.'</code>	
<code>'Auch ein Backslash (\\) muss gequotet werden.'</code>	
<code>'Hallo\\n'</code>	# Hier kein Zeilenumbruch!
<code>'Hallo Welt!'</code>	# Hallo, Zeilenumbruch, Welt! (=11 Zeichen)

Strings in doppelten Anführungszeichen

- Escape-Zeichenfolgen werden interpretiert
- Variablen werden interpretiert („Variablen-Interpolation“) - mehr dazu später

"Hallo"	# das Gleiche wie 'Hallo'
"Hallo\nWelt!"	# Hallo, Zeilenumbruch, Welt! (=11 Zeichen)
"Hallo\tWelt!"	# Hallo, Tabulatorschritt, Welt! (=11 Zeichen)
"Das \"-Zeichen und \\ müssen gequotet werden.“	
"Hier kommt ein \007Pieps. "	# ASCII, oktal
"Hier kommt ein \x45. "	# ASCII, hexadezimal (hier „E“)

- | | | |
|-----------------------------|--|-------------------------------|
| - Addition | - 2 + 3 | |
| - Subtraktion | - 2.4 - 3.7 | |
| - Multiplikation | - 3 * 12 | |
| - Division | - 10.2 / 0.3 | |
| - Restwert (Modulo) | - 10 % 3 10.7 % 3.2 (= 10 % 3) | |
| | - -10.9 % 3.2 (= -10 % 3 = 2) | |
| - Potenzierung | - 2**3 | |
| - Stringverkettung | - "Hallo" . "Welt" | # "HalloWelt" |
| | - "Hallo" . " " . "Welt" | # "Hallo Welt" |
| | - 'Hallo \n' . "\nWelt" | # "Hallo \n" + Umbruch |
| - Stringwiederholung | - "abc" x 3 | # "abcabcabc" |
| | - "abc" x (2+1) | # "abcabcabc" |
| | - 5 x 4 | # = "5" x 4 = "5555" |

Bei Bedarf wandelt Perl Strings in Zahlen (und umgekehrt) automatisch um

- Umwandlungsbedarf wird anhand des Operators bestimmt
- Bei der Konvertierung in Zahlen werden führende Leerzeichen und nachgestellte Nicht-Zahlenzeichen ignoriert

"12" * "3"	# = 36
"12Hallo34" * "3"	# = 36
"12Hallo34" * " 3"	# = 36
"12Hallo34" * " 3 Hallo 2 "	# = 36
"Z" . 5 * 7	# = "Z" . 35 = „Z35“

- Behälter für einen einzelnen skalaren Wert
- Variablen-Name: **Dollar-Zeichen (\$)** + **Perl-Identifizier**
- **Perl-Identifizier:**
 - + Nur alphanumerische Zeichen und Unterstrich (__) erlaubt,
 - + nicht jedoch mit einer Ziffer beginnend
- Es wird zwischen Groß- und Kleinschreibung unterschieden!
(\$test != \$Test)

- **Skalare Zuweisung:**

```
$hello = 'Hallo';  
$pi = 3.14159;  
$hello = $pi * 2;      # Wechsel des Typs  
$hello = $hello - 1.23;
```

- **Binäre Zuweisung:**

```
$hello = 0;  
$hello += 5;           # entspricht "$hello = $hello + 5"  
$hello *= 3;  
$hello **= 2;  
$hello .= "Hallo";
```

- **Autoinkrement und Autodekrement:**

```
$x = $zahl++;  
$y = $zahl--;
```

- **Präinkrement und Prädekrement:**

```
$x = ++$zahl;  
$y = --$zahl;
```

- Bei **Strings in doppelten Anführungszeichen** werden Namen von Variablen durch ihren Wert ersetzt

```
$pi = 3.14159;  
$hello = "Die Zahl PI ist $pi.\n";  
print $hello;  
$hello = 'Die Zahl PI ist $pi.\n';  
print $hello;  
print 'PI ist ' . $pi;  
print "PI ist \"$pi.\"";
```

- Perl sucht nach dem längsten möglichen Variablennamen

```
$ding = 'Auto';  
$anz = 3;  
print "Ich habe $anz $dings.\n";  
print "Ich habe $anz ${ding}s.\n";  
print "Ich habe $anz $ding" . "s.\n";  
print "Ich habe $anz " . $ding . "s.\n";
```

```
$user = 'Erik';  
print "Hallo $usar, schön, Dich zu sehen!\n";  
    Tippfehler!
```

- Die Variable **\$usar** wird automatisch erzeugt („zum Leben erweckt“ - „vivification“) und ist leer

```
Hallo , schön, Dich zu sehen!
```

- Das ***strict-Pragma*** vermeidet solche Probleme
- Variablen müssen dann explizit mit **my** deklariert werden

```
use strict;  
my $user = 'Erik';  
print "Hallo $usar, schön, Dich zu sehen!\n";
```

**Global symbol "\$usar" requires explicit package name at - line 3.
Execution of - aborted due to compilation errors.**

Bedingte Ausführung (auch: Verzweigung, Auswahl, Fallunterscheidung)	if else elsif unless	(einseitige) Auswahl Alternative Weiterführende Prüfung Negierte Auswahl
Schleifen	while until do for foreach	Schleife mit Laufbedingung Schleife mit Abbruchbedingung Fußgesteuerte Schleife Zählschleife Listen-/Arrayschleife
Sprünge	goto next redo last continue	Sprung Nächster Schleifendurchlauf Wiederholung des Schleifendurchlaufes Schleife abbrechen Block hinter einer Schleife
Subroutinen	sub	Deklaration einer Unteroutine

Vergleich	Zahlen	Strings
Gleich	<code>==</code>	<code>eq</code>
Ungleich	<code>!=</code>	<code>ne</code>
Kleiner als	<code><</code>	<code>lt</code>
Größer als	<code>></code>	<code>gt</code>
Kleiner als oder gleich	<code><=</code>	<code>le</code>
Größer als oder gleich	<code>>=</code>	<code>ge</code>

- Gleichheitstest (`==`) **nicht** verwechseln mit Zuweisungsoperator (`=`)!
- Ungleichoperator ist `!=`, **nicht** `<>` (Diamantoperator)!
- Im ASCII-Zeichensatz stehen Großbuchstaben **vor** Kleinbuchstaben!

```

35 != 30 + 5      # falsch
35 == 35.0        # wahr
'35' eq '35.0'    # falsch, weil Stringvergleich
'Hallo' eq "Hallo" # wahr
'hallo' eq 'Hallo' # falsch
"Hallo" gt "hallo" # falsch, „H“ kommt vor „h“
' ' gt " "        # wahr
  
```


- Einfache Auswahl

```
if ($zahl < 0) {  
    print "Die Zahl ist negativ.\n";  
}
```

- Mehrere Statements

```
if ($zahl <= 0)  
{  
    print "Die Zahl ist wahrscheinlich negativ.\n";  
    print "Vielleicht ist sie aber auch 0.\n";  
}
```

**Bedingter Code muss immer in
Schweifklammern stehen
(vgl. jedoch C!)**

- Alternative mit else

```
if ($zahl < 0)  
{  
    print "Die Zahl ist negativ.\n";  
}  
else  
{  
    print "Die Zahl ist 0 oder positiv.\n";  
}
```

- Schachtelung

```
if ($zahl == 0)
{
    print "Die Zahl ist 0.\n";
} else
{
    if ($zahl < 0) {
        print "Die Zahl ist negativ.\n";
    } else {
        print "Die Zahl ist positiv.\n";
    }
}
```

- Weiterführende Prüfung

```
if ($zahl < 0) {
    print "Die Zahl ist negativ.\n"; }
elsif ($zahl == 0) {
    print "Die Zahl ist 0.\n"; }
else {
    print "Die Zahl ist positiv.\n"; }
```

- Kopfgesteuerte Schleife

```
$zaehler = 0;
while ($zaehler <= 10)
{
    $zaehler += 2;
    print "Der Zähler steht jetzt auf $zaehler.\n";
}
# Ausgabe: 2, 4, 6, 8, 10 und 12
```

- Fußgesteuerte Schleife

```
$zaehler = 12;
do
{
    $zaehler += 2;
    print "Der Zähler steht jetzt auf $zaehler.\n";
}
while ($zaehler <= 10); ← Semikolon beachten!
# Ausgabe: 14 !!
```

- **Aufsteigende Zählschleife**

```
for ($i = 1; $i <= 10; $i++)  
{  
    print "Die Variable i hat den Wert $i.\n";  
}
```

- **Absteigende Zählschleife**

```
for ($i = 10; $i >= 1; $i--)  
{  
    print "Die Variable i hat den Wert $i.\n";  
}
```

- **Dreier-Sprünge**

```
for ($i = 1; $i < 10; $i += 3)  
{  
    print "Die Variable i hat den Wert $i.\n";  
}  
# Ausgabe: 1, 4 und 7
```

Der Wert einer nicht-definierten, leeren Variable

- Wird bei Nutzung als Zahl wie 0 behandelt
- Wird bei Nutzung als String wie "" (Leerstring) behandelt
- Dennoch ist undef weder Zahl noch String, sondern eine eigenständige Art eines skalaren Wertes!
- Explizite Zuweisung ist möglich: `$var = undef;`

```
for ($n = 1; $n < 10; $n += 2)
{
    $summe += $n;      # Erster Durchlauf: $summe ist undef!
    $text .= "bla";    # Auch hier ist $text zunächst undef
}
print "Summe: $summe, Text: $text\n";
# Berechnet 1 + 3 + 5 + 7 + 9 = 25
# Summe: 25, Text: blablablabla
```

– Funktion zur Ermittlung, ob eine Variable definiert ist

```
for ($n = 1; $n < 10; $n += 2)
{
    $summe += $n;
    if (defined($text))
    {
        $text .= " + $n";
    }
    else
    {
        $text = $n;
    }
}
print "$text = $summe\n";
# 1 + 3 + 5 + 7 + 9 = 25
```

Start des Perl-Interpreters mit -w-Option:

perl -w meinprogramm.pl

- Im Programm einstellen:

#!/usr/bin/perl -w (Nicht-Unix-Systeme: #!perl -w)

- Per Pragma (ab Version 5.6):

use warnings;

```
use warnings;  
print "Leere Variable: $etwas.\n";  
$x = "12hallo34";  
$x += 3;
```

Name "main::\$etwas" used only once: possible typo at warnings.pl line 3.

Use of uninitialized value in concatenation (.) or string at warnings.pl line 3.

Argument "12hallo34" isn't numeric in addition (+) at warnings.pl line 5.

Wahrheitswerte (wahr/falsch, true/false)

- Eine **Zahl** gilt bei **0** als **falsch**, alle anderen Zahlen sind wahr
- Ein **leerer String (")** gilt als **falsch**, alle anderen Strings sind wahr
- **Achtung:** Der String '0' gilt als Zahl 0, ist somit falsch!
- Umkehrung eines Wahrheitswertes: **Nicht-Operator (!)**

```
$name = "Erik";  
if ($name)  
{  
    print "Zumindest kein Leerstring...\n";  
}  
$zahl = 42;  
if ($zahl)  
{  
    print "Ich weiß, daß die Zahl nicht 0 ist!\n";  
}  
$ist_groesser = $name gt 'Erik';  
if (!$ist_groesser)  
{  
    print "$name ist NICHT größer als 'Erik'. Vielleicht aber gleich!\n";  
}
```