

PERL

A language by Larry Wall

Practical Extraction and Report Language

or

Pathologically Eclectic Rubbish Lister

PERL

- ❑ Entwickelt bei Larry Wall (späten 80) als Ersatz für awk
- ❑ Mittlerweilen der Ersatz für:
 - awk, sed, grep, andere Filter, shell Skripte, C Programme, ...
- ❑ Sehr nützlich weil:
 - Läuft unter Unix, Linux, Mac, Amiga, OS/2, VMS, DOS, Windows, ...
 - Ist der de facto Standard für komplexe Skriptaufgaben
 - Hat Standardbibliotheken für viele Anwendungen
 - Web/CGI, Datenbanken, Sockets, ...

Perl (2.)

- ❑ Eine interpretierte Sprache, die fast so aussieht wie C mit eingebautem sed, awk, und sh , und ein paar Teile von csh, Pascal, FORTRAN, BASIC-PLUS dazu
- ❑ Hoch optimiert zur Manipulation von druckbaren Text (kann aber auch mit binär Daten umgehen)
- ❑ Besonders nützlich für Systemmanagement Aufgaben, wegen Schnittstellen für die meisten Systemaufrufe
- ❑ Reich genug für die meisten Programmieraufgaben
- ❑ „Eine Shell für C Programmierer“ [Larry Wall]

Perl (3.)

- Einige der Kriterien für das Sprachdesign für Perl:
 - Mach es einfach/prägnant häufige Redensarten zu benutzen
 - Stelle viele verschiedene Wege zur Verfügung das selbe zu sagen
 - Benutze sinnvolle Defaultregeln um die Anzahl der Deklarationen zu minimieren
 - Habe keine Angst davor Kontext als syntaktisches Tool einzusetzen
 - Eine große Sprache von der die Nutzer ein Teilmenge lernen werden
- Wie funktioniert das Alles?
 - Eine Sprache, die es einfach macht nützliche Systeme zu implementieren
 - Lesbarkeit kann manchmal ein Problem sein
 - Interpretiert -> Effizienz kann ein Anliegen sein

Beispiel

- Hier ist ein Beispiel Perl Programm
 - Bitte nicht verwirren lassen!

```
while (<>) {  
    next if /^#/;  
    ($x, $y, $z) = /(\S+)\s+(\d\d\d)\s+(foo|bar)/;  
    $x =~ tr/a-z/A-Z/;  
    $seen{$x}++;  
    $z =~ s/foo/fear/ && $scared++;  
    printf "%s %08x %-10s\n", $z, $y, $x  
    if $seen{$x} > $y;  
}
```

Benutzen von Perl

- Einfaches Beispielprogramm

```
print "Hello, world\n";
```

- Kann ausgeführt werden über ein Kommandozeilen Argument zu Perl

```
$ perl -e 'print "Hello, world\n";'
```

```
Hello, world
```

```
$
```

- Wenn es in einer Datei hello steht, kann es auf folgende Art ausgeführt werden

```
$ perl hello
```

```
Hello, world
```

```
$
```

Benutzen von perl (2.)

- Alternativ kann man auch die folgende Zeile in die Datei hello aufnehmen

```
#!/usr/local/bin/perl  
print "Hello, world\n";
```

- Und es ausführen über

```
$ chmod +x ./hello  
$ ./hello  
Hello, world  
$
```

- Für Alternativen zu \#! siehe die Perl Webseiten

Syntaktische Konventionen

- ❑ Als C/C++ Programmierer wird man des öfteren schreiben:

`x = 1;` anstelle von `$x=1;`

- ❑ Perl wird mit folgender Nachricht antworten:

Can't modify constant item in scalar assignment ...

- ❑ Die Fehlermeldung gibt immer die Zeilennummer an (z.B: XYZ)
- ❑ Lösung: gehe zu Zeile XYZ und füge das Dollarzeichen ein

Syntaktische Konventionen

- Perl benutzt nicht-alphabetische Zeichen um verschiedene Arten von Programm Eigenheiten einzuführen (setzt den Kontext in dem die Bezeichnung zu interpretieren ist)

Char	Art	Beispiel	Beschreibung
#	Kommentar	# comment	Rest der Zeile ist Kommentar
\$	Skalar	\$count	Variable mit einfachen Wert
@	Array	@counts	Liste von Werten, indiziert über Integer
%	Hash	%marks	Gruppe von Werten, indiziert über Strings
<	Handle	<STDIN>	Eingabe auf Datei
&	Subroutine	&doIt	Ein aufrufbares Stück Perl Code (Funktion)

- Nicht markierte Bezeichnungen sind entweder Kommandonamen oder Kontrollstrukturen

Variablen

- Perl unterstützt drei Arten von Variablen
 - **Skalare** ... Atomarer Wert (Zahl oder String)
 - **Arrays** ... Werteliste, indiziert bei Zahlen
 - **Hashes** ... Wertegruppe, indiziert bei Strings
- Variablen müssen weder deklariert noch initialisiert werden
- Wenn sie ohne Initialisierung verwandt werden, ist der Wert 0 oder leerer String oder leere Liste
- Vorsicht: Schreibfehler in Variablennamen
 - `print "abc=$ac\n";` anstelle von `print "abc=$abc\n";`
 - Was ist mit Wert von `$abc` geschehen

Variablen (2.)

- ❑ Viele Skalar Operationen haben die Idee einer Default Quelle, Ziel (source/target)
 - ❑ Wenn kein Argument angegeben ist, wird **\$_** genommen!
 - ❑ Vorteil:
 - Oft sehr praktisch um kurze Programme zu schreiben
 - ❑ Nachteil:
 - Verwirrend für neue Benutzer
- (Die Benutzung von **\$_** entspricht der Benutzung von „es“ in der Sprache)

Skalare

- Vier Arten von Datentypen: String, Integer, Float, Boolean
- Werte von skalaren Variablen werden automatisch interpretiert entsprechend dem Kontext (bestimmt durch den entsprechenden Operator)

- Beispiele:

<code>\$y = "123 ";</code>	<code># \$x wird String „123 “ zugewiesen</code>
<code>\$z = 123;</code>	<code># \$z wird Wert 123 zugewiesen</code>
<code>\$i = \$x + 1;</code>	<code># \$x Wert wird interpretiert als integer</code>
<code>\$j = \$y + \$z;</code>	<code># \$y Wert wird interpretiert als integer</code>
<code>\$a = \$x == \$y;</code>	<code># vergleiche \$x, \$y numerisch</code>
<code>\$b = \$x eq \$y;</code>	<code># vergleiche \$x, \$y als Strings</code>
<code>\$c = \$x . \$y;</code>	<code># Konkatenation der Strings \$x, \$y</code>

Konstanten

- Literale Konstanten haben eine Syntax ~ wie in C/C++
- Zwei Arten Strings: "..." (Interpretiert), '...' nicht interpretiert
- Beispiele:

<code>\$answer = 42;</code>	# Ein Integer (oder gegebenenfalls Float)
<code>\$pet = "Camel";</code>	# string
<code>\$msg = "Ich liebe \$pet";</code>	# string mit Interpolation
<code>\$msg = "Gehe \${pet}s";</code>	# string mit Interpolation
<code>\$cost = `Es kostet \$100`;</code>	# string ohne Interpolation
<code>\$dst = \$src;</code>	# Kopieren von Variablen
<code>\$x = \$y + 5;</code>	# Ausdruck
<code>\$cwd = `pwd`;</code>	# string Ausgabe von Kommando
<code>\$stat = system("ls \$d");</code>	# numerischer Status des Kommandos

Konstanten (2.)

- Interpretierte Strings:
 - Interpolation des Wert von jeder Variable im String (z.B: \$x)
 - Interpretiert den Standard Schrägstrich Escapes (\n \t \")
- Nicht interpretierte Strings:
 - Bearbeiten \$ etc als normale Zeichen
 - Erkennen \` um eingebettete Zeichen zu erlauben
- Beispiele:
 - \$x = 1; \$y = "xyz"; \$z = 'abc';
 - \$a = "why isn't \$x better\nthan \"\$y\" or \"\$z\"";
 - \$b = `why isn't \$x better\nthan \"\$y\" or \"\$z`;

a:

why isn't 1 better
than "xyz" or \$z

b:

why isn't \$x better
than \"\$y\" or \"\$z

Array (Listen)

- Ein Array ist eine Sequenz von Skalaren, indiziert über Position (0, 1, 2,)
- Der ganze Array wird mit `@array` beschrieben
- Individuelle Element mit: `$array[index]`
- `$#array` gibt den Index des letzten Elements
- Beispiel:

```
$a[0]="first string"; $a[1] = "2nd string"; $a[2] = 123;
```

Oder

```
@a = ("first string", "2nd string", 123);  
print "Index des letzten Elements ist $#a\n";  
print "Anzahl der Elemente ist ", $#a+1, "\n";
```

Array (Listen) (2.)

- ❑ Arrays müssen nicht deklariert werden
- ❑ Arrays wachsen und schrumpfen nach Bedarf
 - `$#h = 99;` `# bildet ein Array mit 99 Elementen`
- ❑ „Fehlende“ Elemente werden interpoliert
 - `$abc[0]= “abc“; $abc[2]= “xyz“;`
 - `# Zugriff auf $abc[1] gibt “” zurück`
- ❑ Zuweisung zu/von ganzen Array möglich
 - `@numbers = (4,12,5,7,2,9);`
 - `($a, $b, $c, $d) = @numbers;`

Array (Listen) (3.)

- Array können elementeweise zugegriffen werden

```
@nums = (23, 95, 33, 42, 17, 87);
```

```
$sum = 0;
```

```
# optional
```

```
for ($i = 0; $i <= $#nums; $i++) {
```

```
    $sum += $nums[$i];
```

```
}
```

Array (Listen) (4.)

- Die Operationen **push** and **pop** agieren auf dem „rechten“ Ende des Arrays

```
                                # Wert von @a
@a = (1, 3, 5);                 # (1, 3, 5)
push (@a, 7);                   # (1, 3, 5, 7)
$x = pop @a;                     # (1, 3, 5)
```

- Andere nützliche Operationen auf Arrays:

```
sort(@a)                        # gibt sortierte Version von @a zurück
reverse(@a)                     # gibt umgekehrte Version von @a zurück
shift(@a)                       # wie push(@a), aber von links
unshift(@a)                     # wie pop(@a), aber von links
```

Vergleichende Operatoren

- Perl benutzt unterschiedliche Operatoren für Strings/Zahlen
 - Notwendig um zu spezifizieren welcher Vergleich ausgeführt werden soll!

Operation	Numerisch	String
Gleich	<code>==</code>	<code>eq</code>
Nicht gleich	<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>
Vergleich	<code><=></code>	<code>cmp</code>

`$a <=> $b`, 0 falls gleich, 1 falls `$a` größer, -1 falls `$b` größer

Logische Operatoren

- Perl hat zwei Sätze von logischen Operatoren
 - Wie C, wie im Englischen
- Der zweite hat sehr niedrigen Vorrang
 - Benutzung zwischen Statements

Operation	Beispiel	Bedeutung
&&	$x \ \&\& \ y$	falsch falls x falsch, sonst y
 	$x \ \ y$	wahr, falls x wahr, sonst y
!	$! \ x$	wahr, falls x nicht wahr, sonst falsch
and	$x \ \text{and} \ y$	falsch falls x falsch, sonst y
or	$x \ \text{or} \ y$	wahr, falls x wahr, sonst y
not	$\text{not} \ x$	wahr, falls x nicht wahr, sonst falsch

Logische Operatoren (2.)

- Beispiel wie man die logischen Operatoren mit Statements verwendet:

```
if (! open (FILE, „myFile“)) {  
    die „Can‘t open myFile“;  
}
```

kann ersetzt werden mit

```
open (FILE, „myFile“) or die „Can‘t open myFile“;
```

Kontrollstrukturen

- **Semikolon** muss jedes Perl Statement terminieren, z.B.:

```
$x = 1;
```

```
print "Hello";
```

- Alle Statements mit Kontrollstruktur müssen **geklammert** werden {}, z.B.:

```
if ($x > 9999) {
```

```
    print "x is big\n";
```

```
}
```

Iteration

- Geschieht über: **while, until, for, foreach**

```
while ( boolExpr ) {  
    statements;  
}
```

```
until ( boolExpr ) {  
    statements;  
}
```

```
for ( init; boolExpr; step ) {  
    statements;  
}
```

```
foreach var ( list ) {  
    statements;  
}
```

Iteration (2.)

□ Beispiel: Berechnung von $pos = k^n$

Methode 1: while

```
$pow = $i = 1;
while ( $i <= $n ) {
    $pow *= $k;
    $i++;
}
```

Methode 2: for

```
$pow = 1;
for ( $i = 1; $i <= $n ; $i++ )
{
    $pow *= $k;
}
```

Methode 3: foreach

```
$pow = 1;
foreach $i ( 1 .. $n ) {
    $pow *= $k;
}
```

Methode 4: Operator

```
$pow = $k ** $n;
```

Iteration (3.)

- ❑ **foreach** benutzt **\$_** falls keine Variable genannt

```
@countdown = (10,9,8,7,6,5,4,3,2,1);  
foreach (@countdown) {           # benutzt $_  
    print;                       # benutzt $_  
    print "\n";  
}
```

- ❑ Oder sogar

```
foreach (10,9,8,7,6,5,4,3,2,1) {print; print "\n"};
```

- ❑ Oder

```
foreach (10,9,8,7,6,5,4,3,2,1) {print "$_\n"};
```

Ein/Ausgabe

- Dateien werden über **handles** angesprochen
- `<Handle>` für eine Eingabedatei bedeutet „lese die nächste Zeile von dieser Datei“

Z.B.: `$line = <STDIN>;`

- ... Speichert die nächste Zeile der Standardeingabe in die Variable `$line`.
- Ausgabedateihandles werden benutzt als erstes Argument im `print` Kommando

Z.B.: `print REPORT "Report for $today\n";`

- ... Schreibt eine Zeile zu der Datei, die zu `REPORT` Handle angeheftet ist

Ein/Ausgabe (2.)

- Beispiel (ein einfaches cat):

```
#!/usr/local/bin/perl  
# Copy stdin nach stdout  
while ($line = <STDIN>) {  
    print $line;  
}
```

- Oder einfacher

```
while (<STDIN>) { print; }  
# oder sogar  
print <>;
```

Ein/Ausgabe (3.)

- Handles kann man explizit an Dateien heften über das **open** Kommando:

```
open(DATA, "< data");           # lese von Datei „data“  
open(RES, "> result");         # schreibe zu Datei „result“  
open(XTRA, ">> stoff");        # hänge an Datei „stoff“ an
```

- Handles kann man an **Pipelines** heften zum Lesen/Schreiben von Unix Kommandos:

```
open(DATE, "/bin/date |");     # lese Ausgabe von date  
open(FEED, "| more");         # sende Ausgabe an more
```

- Öffnen einer Handle kann fehlschlagen

```
open(DATA, "< data");           or die "Can't open data file";
```

- Handles werden über **close(HandleName)** geschlossen

Ein/Ausgabe (4.)

- ❑ Die spezielle Dateihandle $\langle \rangle$
 - Behandelt alle Kommandozeilenargumente als Dateinamen
 - Öffnet und liest jede von Ihnen
- ❑ Falls es keine Kommandozeilenargument gibt, dann: $\langle \rangle == \langle \text{STDIN} \rangle$
- ❑ Beispiel:

```
perl -e 'print <>;' a b c
```
- ❑ Zeigt den Inhalt der Dateien a, b, und c auf stdout an

String Funktionen

- ❑ Entfernen des letzten Zeichen: **chop**
- ❑ Entfernen des Zeilentrennzeichen: **chomp**
- ❑ Beispiele:

```
chomp($host = `hostname`);
```

```
while (<STDIN>) {  
    chop;  
    ....  
}
```

Assoziative Arrays (Hashes)

- **Hash**: Arrays indiziert über Strings
- Konzeptionell ist ein Hash eine Menge (nicht Liste) von (Schlüssel, Wert) Paaren ((key,) value)
- Man kann ein gesamtes Hash zugreifen über **%hashName**, z.B.:

	# Schlüssel	Wert
%days = (“Sun“	=> “Sunday“,
	“Mon“	=> “Monday“,
	“Tue“	=> “Tuesday“,
	“Wed“	=> “Wednesday“,
	“Thu“	=> “Thursday“,
	“Fri“	=> “Friday“,
	“Sat“	=> “Saturday“);

Assoziative Arrays (Hashes) (2.)

- Individuelle Komponenten des Hashes werden über `$hashName{keyString}` gelesen

- Beispiel:

```
$days{"Sun"};      # gibt „Sunday“ zurück
```

```
$days{"Fri"};     # gibt „Friday“ zurück
```

```
$days{"dog"};    # gibt „“ zurück
```

```
$days{0};        # gibt „“ zurück
```

```
# Einfügen eines neuen Elements
```

```
$days{"dog"} = "Dog Day Afternoon";
```

```
# Ersetzen des Wertes für Schlüssel „Sun“
```

```
$days{"Sun"} = "Soonday";
```

Assoziative Arrays (Hashes) (3.)

- Betrachte die folgenden zwei Zuweisungen:

```
@f = ("John", "blue", "Anne", "red", "Tim", "green");
```

```
%g = ("John", "blue", "Anne", "red", "Tim", "green");
```

- Die erste gibt einen Array von Strings: `$f[0]`
- Die zweite gibt eine Nachschlagtabelle von Namen auf Farben: `$g{"Tim"}`

Assoziative Arrays (Hashes) (4.)

- Um die (Schlüssel, Werte) Paare zu inspizieren

```
foreach $key (keys %myHash) {  
    print “($key, $myHashs{$key})\n“; }  
}
```

Oder für die Werte ohne Schlüssel

```
foreach $val (values %myHash) {  
    print “(?, $val)\n“; }  
}
```

Assoziative Arrays (Hashes) (5.)

- Beispiel (Sammeln von Noten pro Student)
 - Die Datendatei sollte aus (Namen, Note) Paaren bestehen, getrennt über Leerzeichen, ein Eintrag pro Zeile
 - Die Ausgabe sollte sein (Name, Notenliste), wobei die Noten über Kommas getrennt sein sollen

```
while (<>) {  
    chomp;  
    ($name, $mark) = split;  
    $marks{$name} .= "$mark,";  
}  
for $name (keys %marks) {  
    chop($marks{$name});  
    print "$name $marks{$name}\n";  
}
```

Assoziative Arrays (Hashes) (6.)

- Die delete Funktion entfernt einen Eintrag oder Einträge von Assoziativen Arrays

Um ein Paar zu entfernen

```
delete $days{"Mon"};    # I don't like Monday
```

Um mehrerer Paare zu entfernen

```
delete $days{"Sat", "Sun", }; # no weekend
```

Oder das ganze Hash

```
undef %days;
```

Perl Reguläre Ausdrücke

- ❑ Weil Perl sehr auf Strings aufbaut, sind reguläre Ausdrücke ein sehr wichtiger Teil der Sprache
- ❑ Sie können benutzt werden in
 - Bedingungen, um zu testen ein Pattern im String gefunden wird, z.B: Testen des Inhalt eines Strings

```
if ($name =~ /[0-9]/){print "name contains digit\n";}
```
 - In Zuweisungen, um zur Modifikation des Wert des Strings, z.B: Umwandlung von McDonald in MacDonald

```
$name =~ s/Mc/Mac/;
```

Perl Reguläre Ausdrücke (2.)

- Ein regulärer Ausdruck ist ein Muster von Zeichen
- Ein normales Zeichen findet sich selber
- Muster können über Operatoren aus anderen Mustern zusammengesetzt werden

ab	findet Zeichenkombination ab
ab yz	findet Zeichen ab oder yz
[0123456789]	findet jede Ziffer
[0-9]	Abkürzung für jede Ziffer
[range]	jedes Zeichen in range
.	findet jedes Zeichen außer \n
^	findet den Zeilenanfang
\$	findet das Zeilenende
\	Escape für das nächste Zeichen

Perl Reguläre Ausdrücke (3.)

- Wiederholungsoperatoren:

`patt*` 0 oder *mehrere* Auftreten von *patt*

`patt+` 1 oder *mehrere* Auftreten von *patt*

`patt?` 0 oder 1 Auftreten von *patt*

`patt{n,m}` zwischen *n* und *m* viele Auftreten von *patt*

- Perl erweitert POSIX reguläre Ausdrücke mit einigen Abkürzungen:

`\d` findet jede Ziffer, d.h. [0-9]

`\D` findet alles was keine Ziffer ist, d.h. [^0-9]

`\w` jedes „Wort“ Zeichen, d.h. [a-zA-Z_0-9]

`\s` jedes Leerzeichen, d.h. [\t\n\r\f]

Perl Reguläre Ausdrücke (4.)

- Die normale Semantik von pattern matching ist „ersten, dann längsten“ Match sucht

- Beispiel:

`/ab+/` findet **ab**abbb nicht ab**bb**abbb oder abbbab**bbb**

Perl Reguläre Ausdrücke (5.)

□ Verschiedene Möglichkeiten:

- Reiner Match:

`m/pattern/[options]` oder `/pattern/[options]`

- Match mit Ersetzen der ersten Instanz:

`s/pattern/replacement/[options]`

- Ersetzen aller Zeichen in Suchliste durch Ersetzungsliste:

`tr/Suchliste/Ersetzungsliste/[options]`

□ Verschiedene Optionen:

`/i` Ignoriere Groß/Kleinschreibung (case-insensitive)

`/g` Globaler Match: alle Matches im String

`/m` Pattern über mehrere Zeilen

`/o` Pattern zur Laufzeit kompilieren

.....

Zugriff auf gefundene Muster

□ Über Klammerung ():

<code>\$nn</code>	Der nnte geklammerten Ausdruck
<code>\$MATCH \$&</code>	String vom match mit letzten Muster

□ Beispiel:

```
if ($name =~ /[vV]on\s+(.*)/) {  
    print "$1 hatte adelige Vorfahren\n";  
}
```

Listen als Strings

- ❑ Erinnerung an die Noten: „54,67,88“ um eine Liste von Noten darzustellen
- ❑ Können wir in eine echte Liste umformen, z.B. um den Mittelwert zu berechnen?
- ❑ Nutze die **split** Operation
 - Syntax: **split /pattern/, String** gibt Liste zurück
- ❑ **join** Operation Konvertierung Liste in String
 - Syntax: **join('char', Liste)** gibt String zurück

Liste als Strings (2.)

□ Beispiel:

```
$marks = "99,67,85,48,77,84";
```

```
@listOfMarks = split /,/ , $marks;
```

```
# weist @listOfMarks (99,67,85,48,77,84) zu
```

```
$sum = 0;
```

```
foreach $m (@listOfMarks) {
```

```
    $sum += $m;
```

```
}
```

```
$newMarks = join ':', @listOfMarks;
```

```
# weist $newMarks (99:67:85:48:77:84) zu
```

Spezielle Variablen

- ❑ Perl definiert eine Reihe von besonderen Variablen mit Informationen über die Ausführungsumgebung
- ❑ Typischerweise einfache Interpunktionszeichen
 - Z.B: \$! \$@ \$# \$\$ \$% ... (Englische Namen vorhanden!)
- ❑ Die \$_ Variable ist besonders wichtig!!!
 - Default Ort für Zuweisung von Returnwerten
 - Default Argument für viele Operationen
- ❑ Gute Nutzung: „schönen, kurzen“ Programmen
- ❑ Schlechte Nutzung: „kryptische“ Programme

Spezielle Variablen (2.)

- \$_** Default Variable
z. B.: für Eingabe und pattern match
(Mustererkennung)
- \$0** Dateiname des laufenden Perlskripts
- \$1** Gefundener String für 1th regexp in Muster
- \$2** Gefundener String für 2th regexp in Muster
-
- \$\$** Prozessnummer des laufenden Perlskripts
- @ARGV** Liste der Kommandozeilenargumente
- %ENV** Tabelle aller Umgebungsvariablen

Spezielle Variablen (3.)

□ Beispiel (echo in Perl):

```
for ($i = 0; $i <= $#ARGV; $i++) {  
    print "$ARGV[$i] "; }  
print "\n";
```

oder

```
foreach (@ARGV) {  
    print "$_ "; }  
print "\n";
```

oder

```
print "@ARGV\n";
```

Funktionsaufrufe

- ❑ Alle Funktionsaufrufe sind Ausdrücke (der zurückgegebene Wert wird oft ignoriert)
- ❑ Notation für Perl Funktionsaufrufe:
`&func (arg1, arg2, ..., argn);`
- ❑ In fast allen Fällen ist das `&` optional:
`func (arg1, arg2, ..., argn);`
- ❑ In fast allen Fällen kann man die Klammern weg lassen:
`func arg1, arg2, ..., argn;`

Funktionen (Subroutinen)

- Parameter werden von Subroutinen in einem speziellen Array `@_` empfangen. Die Werte können in lokale Variablen kopiert werden

```
$result = &simple($alpha, $beta, @tutti);
```

```
sub simple {  
    my ($x, $y, @rest) = @_  
    my ($sum, %seen);  
    return $sum;  
}
```

- Subroutinen können indirekt aufgerufen werden: `&$foo(@list)`

Packages (Module)

- ❑ Jeder hat sein eigenen Namenbereich
- ❑ Zugriff auf Variablen in einem anderen Package über
`$Package::Variable;`
- ❑ Default Package ist `$main`
- ❑ Das Benutzen von Packages aus anderen Dateien ist möglich über
`use Module;`
- ❑ Nützlich für Abstrakte Datentypen

Referenzen

- Einfachste Abstraktion: zu benutzen wie in C & (Adresse von) Operator:

`$scalarref = \ $foo;`

`$arrayref = \ @ARGV;`

`$hashref = \ %ENV;`

`$coderef = \ &handler;`

`$refref = \ $scalarref;`

- Perl hat Referenzzähler und das Objekt wird freigegeben, wenn der Referenzzähler auf Null geht
- Referenzen haben Typen - Benutze `ref($ref)` um herauszufinden auf welchen Type gezeigt wird

Referenzen zu anonymen Objekten

□ Beispiel Array:

```
$arrayref = [1, 2, ['a', 'b', 'c']];
```

```
$arrayref -> [2][1] == 'b';
```

□ Beispiel Hash:

```
$hashref = {
```

```
    'Adam' => 'Eve',
```

```
    'Clyde' => 'Bonnie'
```

```
}
```

Dereferenzierung

- Der Wert ergibt sich über:

```
$bar = $$scalarref;
```

```
push (@$arrayref, $filename);
```

```
$$arrayref[0] = "January";
```

- Oder

```
$arrayref->[0] = "January";
```

```
$hashref ->{"Key"} = "VALUE";
```

Objekte

- ❑ Ein Objekt ist eine Datenstruktur mit Methoden. Diese Methoden können über eine Referenz aufgerufen werden.
- ❑ Genaueres findet man unter der Perl Dokumentation
- ❑ Beispiel: FileHandle

```
use FileHandle;
```

```
$handle=new FileHandle;
```

```
$handle->open(">logfile") or die "no logfile?";
```

```
$handle->print("logfile opened\n");
```

```
$handle->close();
```

Kommandozeilen Optionen

- Die folgenden sind einige der interessanten Switches von Perl
- v Ausgabe der Versionsnummer
- w Generation von Warnungen für Fehleranfällige Konstrukte
- d Ausführen des Skripts im Debugger
- e wie sed: Perl Skript in der Kommandozeile
- n Schleife über die Eingabe
- p wie -n aber mit Ausgabe jeder Zeile
- a Einschalten von Autosplit in Array @F
- i Editieren von Dateien am Platz

Kommandozeilen Optionen (2.)

□ Beispiele:

Ausgabe der momentanen Version

```
perl -v
```

einfachstes Perl Programm

```
perl -e 'print "hello, world. \n";'
```

löschen von Files gefunden über „find foo -print“

```
perl -ne 'chop; unlink;'
```

addiere erste und letzte Spalte (Filter)

```
perl -ane 'print $F[0] + $F[$#F] . "\n";'
```

In Platz editieren von *.c Dateien: Änderung von foo nach bar

```
perl -pie 's/\bfoo\b/bar/g;' *.c
```

Laufen lassen eines Skripts im Debugger mit und ohne Warnungen

```
perl -d myscript
```

```
perl -w myscript
```

Selektion

- Wird ausgeführt über **if ... elsif ... else**

```
if ( boolExpr1 ) {  
    Statements 1;  
} elsif ( boolExpr2 ) {  
    Statements 2;  
} ...  
else {  
    Statements n;  
}
```

- Es gibt kein switch/case

Selektion (2.)

- **if** kann als Operator benutzt werden

```
if ( $x < 0 ) {  
    print "X ist negative";  
}
```

kann geschrieben werden als

```
print "X ist negative" if ( $x < 0 );
```

oder als

```
print "X ist negative" unless ( $x >= 0 );
```

Dateitestoperatoren

- Perl hat ein umfassende Menge von Operatoren zum Anfragen von Dateiinformationen
 - r, -w, -x Datei ist lesbar, schreibbar, ausführbar
- Benutzt zum Nachprüfen von E/A Operationen
 - r "dataFile" && open DATA, "<dataFile";

Literatur Empfehlungen

□ Für den Anfang

- Programming Perl (3rd Ed)
Larry Wall, Tom Christiansen, Jon Orwant
O'Reilly & Associates, 2000, ISBN:0596000278
- Learning Perl
Randal L. Schwartz, Tom Christiansen
O'Reilly & Associates, 1997; ISBN: 1565922840
- A Little Book on Perl
Robert W. Sebesta
Prentice Hall, 2000; ISBN: 0139279555

Literatur Empfehlungen (2.)

□ Für den Ernsthafte

- Perl Cookbook
Tom Christiansen, Nathan Torkington, Larry Wall
O'Reilly & Associates, 1998; ISBN: 1565922433
- Mastering Algorithms With Perl
Jon Orwant, Jarkko Hietaniemi, John MacDonald, John Orwant
O'Reilly & Associates, 1999; ISBN: 1565923987
- www.perl.com
von verschiedenen Autoren, on-line Aufbewahrungsort für Perl Informationen
- www.cpan.org
von verschiedenen Autoren, übergreifendes Perl Archiv Netzwerk

Wo kann man es finden?

- ❑ In den gnu Archiven
- ❑ Beta Versionen, Perl Packages, etc., siehe CPAN
 - www.cpan.org
- ❑ Dokumentation
 - Unix man Seiten „man perl“
 - HTML
 - Latex
 - Postscript
 - On-line „**man perldoc**“
 - Quick Reference Guid von Johan Vromans
 - Perl FAQ und Meta FAQ
 - comp.lang.perl.misc