Vorlesung

Systemprogrammierung mit Perl

Prof. Dr. G. Raffius

Der Perl Debugger

Kontrolle der Script Ausführung	
T	Stack trace
s [expr]	Single Step [in expr]
n [expr]	Next, springt über Unterprogramme
<CR/Enter>	Wiederhole letztes n oder s
r	Return from subroutine
c [ln|sub]	Continue until position
L	List break/watch/actions
t [expr]	Toggle Trace [trace expr]
b [ln|event|sub] [cnd]	Setze breakpoint
B ln|*	Lösche einen/alle Breakpoints
a [ln] cmd	Führe cmd vor Zeile aus
A ln|*	Lösche eine/alle Aktionen
w expr	Füge einen Watch Ausdruck hinzu
W expr|*	Lösche einen/alle Watch Ausdrücke
![!] syscmd	Führe syscmd in einem Subprocess aus
R	Versuche einen Neustart
q or ^D	Quit

Perl Debugger: Daten Untersuchung

Daten Untersuchung	
expr	Führe Perl Codeaus, siehe auch: s,n,t expr
x|m expr	Evaluiert expr in einem Listen Kontext, druckt das Ergebnis oder listet Methoden.
p expr	print expression (Benutzt script's augenblicklichliches package).
S [[!]pat]	Listet Subroutine Namen die [nicht] mit dem Muster übereinstimmen
V [Pk [Vars]]	Listet Variablen in dem Package. Vars kann sein: ~pattern oder !pattern.
X [Vars]	Das gleiche wie "V current_package [Vars]".
y [n [Vars]]	Listet Lexikalische Variablen in höherem scope <n>. Vars ist das gleiche wie bei V.

Hochschule Darmstadt
Fachbereich Informatik

Systemprogrammierung mit Perl

1. Einführung

Einführung

- Perl wurde erfunden von Larry Wall einer Internetlegende
- Perl war anfangs gedacht als "Glue Language".
- Die Ursprünge von Perl liegen in den Tools "sed" dem Streamlineeditor von Unix, "awk" einer Mustererkennungs Sprache und ind en Sprachen C, Pascal, Basic und der Unix Shell, die die Syntax und einzelne Sprachelemente beigesteuert haben.
- Die Version 1.0 von Perl wurde am 18. Dezember 1987 veröffentlicht und wurde seitdem von vielen Leuten auf der Welt als öffentliches Projekt weiterentwickelt.
- Seit der Version 5 ist Perl eine vollständige Programmiersprache mit Objekten
- Die aktuelle Version von Perl ist 5.8.8
- Die nächste große Änderung kommt mit Perl 6, das im Moment in Entwicklung ist und noch einige Zeit brauchen wird, bis es fertig ist

Warum Perl

- Perl ist sehr einfach zu erlernen
- Perl wurde entwickelt, dass es einfach ist für Menschen zu schreiben und nicht dass es einfach ist für Computer zu lesen. Die Sprache ist flexibel wie eine natürliche Sprache und es gilt das Motto "There's More Than One Way To Do It"
- Perl ist sehr portabel und läuft auf fast jedem Computersyste. Die Programme können meist ohne Änderung von einem Betriebssystem zum nächsten übernommen werden
- Perl versteht Texte. Perl Programme handeln mit Wörtern und Sätzen und Paragraphen und nicht mit Arrays von Charaktern
- Perl ist eine echte Hochsprache. Der Anwender muss sich nicht um die Reservierung von Speicher oder die Länge von Arrays kümmern.
- Perl ist frei. Die Sprache wird unter der GPL und unter der Artisitc Licence verteilt
- Perl besitzt eine sehr große öffentliche Bibliothek mit Modulen (CPAN)

Interpretierte / Compilierte Sprachen

- Perl ist wie Java eine Sprache die in einen Zwischencode compiliert und dann interpretiert wird.
- Durch die Interpretation des Zwischencodes ist wie bei Java eine hohe Portabilität gewährleistet
- Viele Funktionen sind intern zur Laufzeiterhöhung in optimiertem C Code geschrieben. Ebenso sind einige der Bibliotheken auf die Perl zurückgreift in C geschrieben
- Bibliotheken werden beim Programmstart mit Hilfe der "use" Anweisung geladen und dann ebenfalls compiliert
- Zur Performanceoptimierung werden Bibliotheken bei modperl im Apache nur beim Programmstart geladen und dann in einem Cache gespeichert
- Nach Aussage von Larry Wall ist Perl um den Faktor π langsamer als ein optimiertes C Programm

Libraries, Module, Packages

- Eine der Stärken von Perl ist das große Angebot von frei verfügbaren Modulen
- Perl besitzt im Internet eine große Bibliothek (www.cpan.org) in der man zu fast jedem Thema schon fertige Module finden kann
- Die übliche Vorgehensweise bei einer Perlentwicklung besteht daher darin, dass man als erstes überlegt was man braucht und dann danach sucht was schon vorhanden ist. Erst danach beginnt man mit der Entwicklung eigener Module.
- Einige wichtige Pakete wie "DBI", "CGI", "LWP", "Net", "SOAP", "Template", "Tk" oder "XML" sind meist schon in der Perldistribution enthalten
- Weitere Pakete können mit dem Programm cpan oder dem PerlPackageManager ppm nachgeladen werden
- Die meisten der Pakete besitzen ein objektorientiertes Interface und definieren für sich einen eigenen Namensraum

Ein erstes Perl Programm

```
#!/usr/bin/perl -w

print "hello world\n";
```

- #! teilt unter Unix der Shell mit, mit welchem Programm das File ausgeführt werden soll
  - ⇨ In unserem Fall wird das Programm /usr/bin/perl benutzt, um das File auszuführen
  - ⇨ -w schaltet die Warnungen des Perl Compilers ein
- print ist ein Befehl in Perl, der eine Liste von Strings erwartet und diese auf einem Ausgabekanal (STDOUT wenn nichts weiter angegeben ist) ausgibt
- "Hello World \n“ ist ein String mit abschließendem newline
- Jedes Perl Statement wird durch ein Semikolon „ ; “ beendet

Etwas verbessertes Hello World Programm

```
#!/usr/bin/perl
use warnings;
use strict;

# Ausgabe einer kurzen Nachricht
print "hello world\n";
```

- use lädt eine Bibliothek
  - ⇨ es wird die Bibliothek warnings.pm geladen, die dafür sorgt, dass der Compiler erweiterte Fehlermeldungen und Warnungen ausgibt (auch unter Windows)
  - ⇨ Die Bibliothek strict.pm wacht darüber, dass einige unsichere Konstrukte nicht benutzt werden
- # leitet einen Kommentar ein
  - ⇨ Alle Zeichen die sich in einer Zeile hinter dem # Symbol befinden werden vom Compiler ignoriert

print

```
#!/usr/bin/perl
use warnings;

print "hello world\n"; # Ausgabe einer kurzen Nachricht
print "Dies ", "ist ", "ein ", "längerer ", "Satz\n“;
print ("Dies ", "ist ", "ein ", "längerer ", "Satz\n“);
print ("Dies ", "ist "), "ein ", "längerer ", "Satz\n“;
```

- Print erwartet als Argument eine Liste von Strings
- Durch den Komma Operator " , " werden die Strings zu einer Liste zusammengefügt
- Eine Liste kann durch runde Klammern gruppiert werden
- In der letzten Ausgabe erhält print als Argument nur die Liste ("Dies ", "ist "). Der Rest wird mit dem Ergebnis von print zu einer neuen Liste kombiniert

Ein Beispiel für Hello World im Internet

```
use CGI qw(:standard );
use strict;

print  header,                    # create the HTTP header
       start_html('hello world'), # start the HTML
       h1('hello world'),         # level 1 header
       end_html;                  # end the HTML
```

- Mit use CGI wird die CGI Bibliothek geladen mit qw(:standard) sagen wir der Bibliothek, dass alle HTML Befehle in den Namensraum des Programms zu importieren sind
  - ⇨ Deshalb können wir jetzt mit h1(..) einen Level 1 Header erzeugen
- Alle Ausgaben werden durch einen print Befehl auf STDOUT erzeugt

Beispiel 2: dynamische Webseiten

```
use CGI qw(:standard *table );
use strict;

print  header;                          # create the HTTP header
print  start_html('hello world');       # start the HTML
print  h1('Inhaltsverzeichnis');        # level 1 header
print start_table;
foreach (qx(ls cgi-bin)) {
      print Tr(td($_));
}
print end_table;
print end_html;                         # end the HTML
```

- Mit qx(...) führen wir einen Shell Befehl aus und erhalten das Ergebnis des Befehls als Returnwert. Statt mit qx(...) kann man auch mit `...` einen Befehl ausführen.
- Dem Kommando ls wurde das Verzeichnis cgi-bin mitübergeben, da ls im Pfad ausgeführt wird, von dem aus der Webserver gestartet wurde
- Die Kommandos start_table und end_table erzeugen in HTML die entsprechenden Tabellentags, Tr(...) einen Zeilentag und td einen Zellentag

Hochschule Darmstadt
Fachbereich Informatik

Systemprogrammierung mit Perl

2. Einfache Daten und Operationen

Einfache Daten

```
#!/usr/bin/perl
use warnings;
use strict;

print "\tThis is a double quoted string\n";
print '\tThis is a double quoted string\n';
print "Some Integer Numbers: ", 7, -48, 65537, 0x23, 25_000_000, "\n";
print "Floating Point Numbers: ", 1.23, 3.14159, 1e3
```

- Literale
  - ⇨ Strings: "Hello World\n", 'Hello Worl\n'
    - Strings in Double Quotes(") werden interpoliert, d.h. Escapesequenzen und Variablen werden erstetzt
    - Strings in Single Quotes(') werden nicht interpoliert
  - ⇨ Integerzahlen: 7, -48, 65537, 0x23, 25_000_000
  - ⇨ Floatingpointzahlen: 1.7, 5e-23

Escape Sequenzen

Ausdruck	Bedeutung
\n	New line, Neue Zeile
\r	Return
\t	Tabulator
\f	Form feed, neue Seite
\b	Backspace
\v	vertkaler Tabulator
\a	Tonsignal
\e	Escape
\007	jeder oktale Wert
\x7f	jeder hexadezimale Wert
\cC	jedes „Control Zeichen“ hier Control-C
\\	Backslash
\“	dopelter Apostroph
\l	nächsten Buchstaben klein schreiben
\L	alle folgenden Buchstaben bis \E klein schreiben
\u	nächsten Buchstaben groß schreiben
\U	alle folgenden Buchstaben bis \E groß schreiben
\E	\L oder \U beenden

Alternative Quoting Formen

```
#!/usr/bin/perl
use warnings;
use strict;

print qq(print "Hello World";\n)
print q(print "Hello World";\n), "\n";
my @days = qw(Montag Dienstag Mittwoch Donnerstag Freitag Samstag Sonntag);
my @files = qx(ls);
```

- Der Operator qq(...) erzeugt einen interpolierten String
  - ⇨ Alternativ kann jedes nicht alphanumerische Zeichen als Begrenzer benutzt werden
    - qq[..], qq{..}, qq/../, qq<..>
- Der Operator q(...) erzeugt einen nicht interpolierten String
  - ⇨ Alternativ kann jedes nicht alphanumerische Zeichen als Begrenzer benutzt werden
- Der Operator qw(...) erzeugt eine Liste von Wörtern
  - ⇨ Trennzeichen sind hier Whitespaces
- Der Operator qx(...) führt einen Shell Befehl aus und liefert das Ergebnis zurück
  - ⇨ Dies ist die alternative Quotingform zu Backticks (`...`)

Here Dokumente

```
#!/usr/bin/perl
use warnings;
use strict;

print <<EOF;

Dies ist ein here Dokument und wird beendet durch ein EOF,
das alleine an erster Stelle in einer Zeile steht

EOF
```

- Here Documente sind wichtig, wenn der String aus einem größeren Textstück mit vorhandenen Umbrüchen besteht.

Operatoren: Zuweisung

```
$i = $i * 5;
$i *= 5;
```

- Die Zuweisung eines Wertes zu einer Variablen geschieht durch das "="-Zeichen.
- Wie in C gibt es in Perl eine verkürzte Schreibweise für Operationen, die den Wert einer Variablen verändern.

Bit-Operatoren

&	bitweises UND
|	bitweises ODER
^	bitweises XOR
~	bitweises Komplement
<<	bitweise Verschiebung des linken Arguments um eine (Ganz-)Zahl nach links (rechtes Argument)
>>	bitweise Verschiebung des linken Arguments um eine (Ganz-)Zahl nach rechts (rechtes Argument)

Logik-Operatoren

!	logisches NOT
&&	logisches UND
||	logisches ODER
not	logisches NOT
and	logisches UND
or	logisches ODER
xor	logisches XOR

Arithmetik Operatoren

+	positives Vorzeichen (unär)
-	negatives Vorzeichen (unär)
+	Addition (binär)
-	Subtraktion (binär)
*	Multiplikation
/	Division
%	Resteiner Division (Modulo)
**	Potenzbildung
++	Inkrement
--	Dekrement

Arithmetische Vergleichsoperatoren

==	liefert true bei Gleichheit
!=	liefert true bei Ungleichheit
>	liefert true, falls linkes Argument größer als rechtes Argument
<	liefert true, falls linkes Argument kleiner als rechtes Argument
>=	liefert true, falls linkes Argument größer oder gleich rechtem Argument
<=	liefert true, falls linkes Argument kleiner oder gleich rechtem Argument
<=>	liefert -1,0,1 je nachdem, ob das linke Argument kleiner, gleich oder größer als das rechte Argument ist

Prioritäten

Assoziativität	Operator
links	Terme (Variablen, geklammerte Ausdrücke, ...)
links	->
-	++ --
rechts	**
rechts	! ~ \ + - (unär)
links	=~ !~
links	* / % x
links	. + - (binär)
links	<< >>
-	unäre Operatoren wie Funktionen mit 1 Argument
-	< > <= >= lt gt le ge
-	== != <=> eq ne cmp

Prioritäten

Assoziativität	Operator
links	&
links	| ^
links	&&
links	||
-	..
rechts	?:
rechts	= += -= *= usw.
links	, =>
-	Listenoperatoren
links	not
links	and
links	or
links	xor

Funktionen für Zahlen

abs($x)	Absolutwert von $x
atan2($x,$y)	Arcustangens von $x/$y (zwischen -pi und +pi)
cos($x)	Cosinus von $x (im Bogenmaß)
exp($x)	Exponentialfunktion ("e hoch $x", wobei e=2,71828...)
log($x)	Natürlicher Logarithmus von $x (Beachte: $x muß positiv sein. Außerdem gilt: $x == log(exp($x)) für alle $x)
sin($x)	Sinus-Funktion von $x (im Bogenmaß)
sqrt($x)	Quadratwurzel aus $x (Beachte: $x muß positiv sein)

Operatoren für Zeichenketten

```
"Hallo" . "Welt"          # identisch mit „HalloWelt“
"Fred" . " " . " Barney"  # das gleiche wie "Fred Barney“

"Fred" x 3          # ist „FredFredFred“
"-" x 20            # ----------
```

- "." Verkettung von Strings
- "x" Vervielfachung von Zeichenketten

Konvertierung zwischen Zahlen und Zeichenketten

```
"X" . (4*5)   # das gleiche wie „X20“
```

- Wird eine Zeichenkette als Operand in einer arithmetischen Operation benutzt, so konvertiert Perl die Zeichenkette automatisch in eine Zahl.
- Leerräume und nichtnumerische Daten werden ignoriert
- Wird ein numerischer Wert in einer Stringoperation benutzt, so wird er automatisch in einen String gewandelt

Vergleiche von Zeichenketten

eq	liefert true bei Gleichheit
ne	liefert true bei Ungleichheit
gt	liefert true, falls linkes Argument größer als rechtes Argument
lt	liefert true, falls linkes Argument kleiner als rechtes Argument
ge	liefert true, falls linkes Argument größer oder gleich rechtem Argument
le	liefert true, falls linkes Argument kleiner oder gleich rechtem Argument
cmp	liefert -1,0,1 je nachdem, ob das linke Argument kleiner gleich oder größer als das rechte Argument ist

Variablen

```
#!/usr/bin/perl
#varint1.pl
use warnings;
use strict;
my $name = "fred";
print "My name is $name\n";
```

- Variablen werden durch das Schlüsselwort "my" deklariert.
- Variablen die nicht unmittelbar initialisiert werden haben den Wert undef
- Skalare Variablen können Strings, Integer, Floatingpointzahlen und Referenzen auf andere Dinge speichern
- Der Name einer Skalaren Variablen beginnt immer mit einem $ Symbol

Nutzung von Variablen

```
#!/usr/bin/perl
#vars3.pl
use warnings;
$a = 6*9;
print "Six nines are ", $a, "\n";
$b = $a + 3;
print "Plus three is ", $b, "\n";
$c = $b / 3;
print "All over three is ", $c, "\n";
$d = $c + 1;
print "Add one is ", $d, "\n";
print "\nThose stages again: ", $a, " ", $b, " ", $c, " ", $d, "\n";
```

- Variablen können Werte zugewiesen werden und sie können in Ausdrücken benutzt werden

Interpolation von Skalaren in Zeichenketten

```
$a = "Fred";
$b = "etwas Text $a" ;          # $b ist jetzt "etwas Text Fred"
$c = "unbekannte Variable $what";          # $c ist "unbekannte Variable"
$d = "etwas Text \U$a";          # $d ist "etwas Text FRED"
$bigbarney = "BARNEY";
$capbarney = "\u\L$bigbarney";          # $capbarney ist "Barney"
```

- In Zeichenketten die in doppelte Apostrophen eingeschlossen sind findet eine Variablenersetzung statt

Interpolation von Skalaren in Zeichenketten

```
#!/usr/bin/perl
#varint4.plx
use warnings;
use strict;
my $times = 8;
print "This is the ${times}th time.\n";
```

- Um Mehrdeutigkeiten zu vermeiden kann der Name der Variablen in geschweifte Klammern gesetzt werden

Die Operatoren chop und chomp

- Der Operator chop entfernt das letzte Zeichen einer Zeichenkette und liefert es als Ergebnis zurück wenn er auf einen String angewandt wird
- Der Operator chomp entfernt alle Zeilenvorschübe am Ende eines Strings und liefert die Anzahl der entfernten Zeichen zurück
  - ⇨ dies ist unabhängig vom Betriebssystem, aber nur wenn der Zeilenvorschub der Betriebssystemskonvention entspricht
  - ⇨ Vorsicht bei Windows Files unter Unix!

wichtige Systemvariablen

Spezial Variablen	
$_	default variable
$0	program name
$/	input separator
$\	output separator
$|	autoflush
$!	sys/libcall error
$@	eval error
$$	process ID
$.	line number
@ARGV	command line args
@INC	include paths
@_	subroutine args
%ENV	environment

<STDIN> als Skalarvariable

```
$a = <STDIN>;          # Text einlesen
chomp ($a);        # Newline löschen

# Kurzfassung
chomp ($a = <STDIN>);
```

- Einlesen von der Konsole erfolgt über den Operator <STDIN>

Beispiel für <STDIN>

```
#!/usr/bin/perl
#currency2.pl
use warnings;
use strict;
print "Currency converter\n\nPlease enter the exchange rate: ";
my $Dollar = <STDIN>;
print "49518 Dollar is ", (49_518/$Dollar), " Euro\n";
print "360 Dollar is ",   (   360/$Dollar), " Euro\n";
print "30510 Dollar is ", (30_510/$Dollar), " Euro\n";
```

Hochschule Darmstadt
Fachbereich Informatik

Systemprogrammierung mit Perl

3. Arrays

Arrays

- Arrays sind grundsätzlich eindimensional (Vektoren) und enthalten als Elemente skalare Größen.
- Die Skalare müssen nicht vom gleichen Typ sein, sondern können gemischt in einem Array auftreten
- Wie bei Zeichenketten muss auch bei Arrays keine Speicherplatzreservierung vorgenommen werden und Arrays können im Laufe eines Programms praktisch beliebige Größen annehmen.
- Versucht man, auf ein nicht definiertes oder gesetztes Element zuzugreifen, so erhält man eine "0" im Zahlenkontext bzw. einen Leerstring "" im Zeichenkettenkontext zurück
  - ⇨ intern besitzt die Variable den Wert "undef"
  - ⇨ Über den Operator defined kann überprüft werden ob die Variable einen definierten Wert besitzt

Array Zugriff

```
@fred = (7,8,9);
$b = $fred[0];          # weist $b den Wert 7 zu
$fred[0] = 5;       # jetzt ist @fred (5,8,9)

$c = $fred[1];
$fred[2]++;         # inkrementieren
$fred[2] += 4;          # addieren
($fred[0], $fred[1]) = ($fred[1], $fred[0]);     # vertauschen
```

- Der Zugriff auf Arrayelemente wird mit dem Subscript Operator durchgeführt.
- Dabei wird der Arrayname nicht mehr durch ein vorangestelltes @ gekennzeichnet sondern durch ein $, da es sich um einen Skalar handelt

Operatoren für Arrays (Zuweisung)

```
@fred = (1,2,3);          # dem Array Fred wird ein Literal zugewiesen
@barney = @fred;          # das Literal wird kopiert und barney
          # zugewiesen
@huh = 1;              # @huh zeigt auf die Liste (1)

@fred = ("eins", "zwei");
@barney = ( 4, 5, @fred, 6, 7);        # @barney wird zu
                       # (4, 5, „eins“, „zwei“, 6, 7)
@barney = ( 8, @barney );       # stellt 8 vor @barney
@barney = (@barney, "aus");   # ein aus an den Schluß
```

Operatoren für Arrays (Zuweisung)

```
($a, $b, $c) = (1, 2, 3);           # $a wird 1, $b wird 2, $c wird 3
($a, $b) = ($b, $a)           # $a und $b werden vertauscht
($d, @fred) = ($a, $b, $c);       # $d wird $a, @fred wird ($b, $c)
($e, @fred) = @fred;        # das erste Element aus @fred wird
                # entfernt und $e zugewiesen

@fred = (4,5,6);              # Initialisierung von @fred
$a = @fred;               # $a erhält den Wert 3, die Länge von
          # @fred

@fred = @barney = (1,2,3);    # der Wert einer Zuweisung ist
                    # selbst wieder ein Array
```

String Zuweisungen

```
#!/usr/bin/perl
# dayarray.plx
use warnings;
use strict;

my @days;
@days = qw(Monday Tuesday Wednesday Thursday Friday Saturday Sunday);
print @days, "\n";
```

Array Initialisierung

- Die Elemente eines Arrays können im wesentlichen auf zwei Arten mit Werten besetzt werden :

```
@vector = (4,6,"ein String",25);
```

  - ⇨ Hier bekommen die ersten vier Elemente von @vector die entsprechenden Werte.
- Bei der Zuweisung über einen Index ist hier das Dollar-Zeichen $ zu beachten, da "vector[7]" eine skalare Variable und kein Array darstellt !

```
$vector[7] = 42;
```

- Die Indizierung der Arrays beginnt normalerweise bei 0.
- Eine weitere Form der Initialisierung ist eine Liste von Wörtern mit dem qw Operator zu bilden

```
@days = qw(Monday Tuesday Wednesday Thursday Friday Saturday Sunday);
```

Array Literale

```
(1,2,3)         # Array mit den drei Werten 1,2,3
(„Fred“, 4.5) # zwei Werte, „Fred“ und 4.5
($b+$c, $d+$e)   # zwei Werte

()          # leeres Array
(1..5)          # das gleiche wie (1,2,3,4,5)
(1.2 .. 5.2)    # das gleiche wie ( 1.2, 2.2, 3.2, 4.2, 5.2 )
(1.2 .. 4.8)    # das gleiche wie ( 1.2, 2.2, 3.2, 4.2)
( 2..6, 10, 12)      # das gleiche wie ( 2, 3, 4, 5, 6, 10, 12)
($a .. $b)      # Eine Liste mit allen Werten die zwischen $a und $b liegen
print („Die Antwort lautet“, $a, „\n“); # Array Literal mit 3 Elementen
```

Array Variablen

```
@fred          # die Arrayvariable @fred
@Ein_sehr_langer_Array-Variablen_Name
```

- Eine Arrayvariable bietet Platz für einen einzelnen Arraywert
- Der Name einer Arrayvariablen beginnt mit einem @ Zeichen
- Eine Arrayvariable die noch nicht initialisiert ist, hat den Wert (), die leere Liste

Slices

```
@fred[0,1]                  # das gleiche wie ($fred[0], $fred[1])
@fred[0,1] = @fred[1,0]         # Elemente vertauschen
@fred[0,1,2] = @fred[1,1,1]     # alle 3 Elemente werden mit dem
                        # 2. Element belegt
@fred[1,2] = (9,10)
@who = ( „fred“, „barney“, „betty“, „wilma“)[2,3]

@fred = ( 7, 8, 9);
@barney = ( 2, 1, 0);
@backfred = @fred[@barney];       # (9,7,8)
```

- Der Zugriff auf eine Liste von Elementen im gleichen Array wird Slice genannt

Die Länge eines Arrays

```
my @array1;
my $scalar1;
@array1 = qw(Monday Tuesday Wednesday Thursday Friday Saturday Sunday);
$scalar1 = @array1;

print "Array 1 ist @array1\nScalar 1 ist $scalar1\n";
print "Die Länge ist ", scalar(@array1), "\n";
```

- Die Länge eines Arrays erhält man, indem man das Array in einem Skalaren Kontext abfragt
- Dies kann entweder durch eine Zuweisung zu einem Skalar erfolgen oder durch Anwendung des scalar Operators

Die Länge eines Arrays

```
my @questions = qw(Java Python Perl C);
my @punchlines = (
        "None. Change it once, and it's the same everywhere.",
        "One. He just stands below the socket and the world revolves around him.",
        "A million. One to change it, the rest to try and do it in fewer lines.",
        '"CHANGE?!!"'
);

for (0..$#questions) {
        print "How many $questions[$_] programmers does it take to change a lightbulb?\n";
        sleep 2;
        print $punchlines[$_], "\n\n";
        sleep 1;
}
```

- Die Länge eines Arrays kann man auch herausfinden indem man den letzten Index des Arrays ermittelt.
- Der letzte Index eines Arrays mit dem Namen @myArray ist über den Skalar $#myArray zugreifbar

For Loop

```
my @array = qw(America Asia Europe Africa);
my $element;
for $element (@array) {
      print $element, "\n";
}
```

- Mit for Loops kann man über ein Array iterieren
- Der Iterator ist eine Alias auf das jeweils iterierte Arrayelement
- Wird kein Iterator angegeben, so wird automatisch $_ mit dem jeweiligen Arrayelement belegt

For Loop

```
my @array=(10, 20, 30, 40);
print "Before: @array\n";
for (@array) { $_ *= 2 }
print "After: @array\n";
```

- Über den Iterator kann das jeweilige Element des Arrays geändert werden
- Der Iterator stellt einen Alias auf das jeweilige Arrayelement dar

Operatoren auf Arrays

- pop
  - ⇨ entferne das letzte Element eines Arrays und liefere es zurück
- push
  - ⇨ hänge ein oder mehrere Elemente an das Ende eines Arrays
- shift
  - ⇨ entferne das erste Element eines Arrays und liefere es zurück
- splice
  - ⇨ füge hinzu oder entferne Elemente irgendwo im Array
- unshift
  - ⇨ Setze ein oder mehrere Elemente an den Anfang eines Arrays

Funktionen für Listen

- grep
  - ⇨ locate elements in a list test true against a given criterion
- join
  - ⇨ join a list into a string using a separator
- map
  - ⇨ apply a change to a list to get back a new list with the changes
- qw/STRING/
  - ⇨ quote a list of words
- reverse
  - ⇨ flip a string or a list
- sort
  - ⇨ sort a list of values
- unpack
  - ⇨ convert binary structure into normal perl variables

Push und Pop

```
sub getPhList
{
   my @list = ();
   my $sth = $dbh->prepare(qq(SELECT *  FROM `personalhandbuch` ORDER BY
                     nachname, vorname));
   $sth->execute();
   while ( my $ref = $sth->fetchrow_hashref()) {
     push @list, $ref;
   }
   return \@list;
}
```

- Mit push kann man Listen aufbauen
- Das Beispiel zeigt eine Datenbankabfrage und den Aufbau einer Liste mit Personennamen
- Zurückgegeben wird eine Referenz auf die aufgebaute Liste

Shift und Unshift

```
sub dividiere
{
      my $dividend = shift @_;
      my $divisor = @_;
      return undef if $divisor == 0;
      return $dividend/$divisor;
}
```

- Die häufigste Anwendung von shift besteht darin die Parameter aus dem Übergabearray @_ auszulesen.

Splice

```
push(@a,$x,$y)      splice(@a,@a,0,$x,$y)
pop(@a)         splice(@a,-1)
shift(@a)       splice(@a,0,1)
unshift(@a,$x,$y)   splice(@a,0,0,$x,$y)
$a[$i] = $y         splice(@a,$i,1,$y)
```

- splice kann Elemente an beliebigen Stellen in einem Array löschen oder hinzufügen
- # splice ARRAY,OFFSET,LENGTH,LIST
- # splice ARRAY,OFFSET,LENGTH
- # splice ARRAY,OFFSET
- # splice ARRAY

Sort

```
my @unsorted = qw(Cohen Clapton Costello Cream Cocteau);
print "Unsorted: @unsorted\n";
my @sorted = sort @unsorted;
print "Sorted:   @sorted\n";
@unsorted = (1, 2, 11, 24, 3, 36, 40, 4);
@sorted = sort @unsorted;
print "Sorted:   @sorted\n";
```

- Der Befehl "sort" sortiert Strings in alphabetischer Reihenfolge
- Zahlen werden bei dieser Form des Aufrufs erst in Strings gewandelt und dann sortiert

sort

```
my @unsorted = (1, 2, 11, 24, 3, 36, 40, 4);

my @string = sort { $a cmp $b } @unsorted;
print "String sort:  @string\n";
my @number = sort { $a <=> $b } @unsorted;
print "Numeric sort:  @number\n";
my @reverse_number = sort { $b <=> $a }  @unsorted;
print "Reverse numeric sort: @reverse_number\n";
```

- Die Steuerung der Sortierung kann durch ein Codefragment in geschweiften Klammern "{...}" erfolgen, das der Funktion sort übergeben wird.
- sort setzt die beiden Variablen $a und $b als Alias von Arrayelementen
- Vorsicht: zwischen geschweiften Klammern und Arrayvariablen steht kein Komma!

Join

```
$rec = join(':', $login,$passwd,$uid,$gid,$gcos,$home,$shell);
print join ", ", @csv_array;        # Erzeugung eines CSV Strings aus einem Array
```

- Mit Join können die Elemente eines Arrays zu einem String verbunden werden.
- Der erste Parameter ist ein Ausdruck, der einen String liefert und zur Verknüpfung der Arrayelemente benutzt wird

Split

```
split /PATTERN/,EXPR,LIMIT
split /PATTERN/,EXPR
split /PATTERN/
split
```

- split zerlegt einen Ausdruck EXPR an den durch Pattern gegebenen Stellen und liefert im Arraykontext die erhaltenen Stücke in einem array zurück.
- Die durch Pattern spezifizierten Stücke werden verworfen, wenn das Muster nicht in Klammern "(...)" gesetzt wird
- Im skalaren Kontext liefert split die Anzahl der Stücke zurück und schreibt das Ergebnis in das Array @_
- Ist kein Ausdruck angegeben, dann zerlegt split den Ausdruck $_
- Ist kein Muster vorgegeben oder das Muster ' ', zerlegt split den Ausdruck in Worte
- Ist der Parameter Limit angegeben, so gibt er die maximale Anzahl von Stücken an, in die der Ausdruck zerlegt wird

Beispiel zu split und join

```
my $passwd = "kake:x:10018:10020::/home/kake:/bin/bash";
my @fields = split /:/, $passwd;
print "Login name : $fields[0]\n";
print "User ID : $fields[2]\n";
print "Home directory : $fields[5]\n";

my $passwd2 = join "#", @fields;
print "Original password : $passwd\n";
print "New password :      $passwd2\n";
```

Grep

```
grep BLOCK LIST
grep EXPR,LIST
```

```
my @ergebnis = grep { $_ > 5 } (1,2,3,4,5,6,7,8,9);
my @muschelmonate = grep /r/, qw( Januar Februar März April Mai Juni Juli August
September Oktober November Dezember);
```

- Die Funktion grep extrahiert aus einer Liste eine Liste
- In der ersten Form wird grep ein Codefragment übergeben. Ist das Ergbnis des Fragments TRUE, dann wird das entsprechende Listenelement in die Ergebnisliste übernommen. Die Variable $_ ist jeweils ein Alias auf das untersuchte Listenelement
- In der zweiten Form wird ein Patternmatching auf das jeweilige Listenelement vorgenommen. Findet ein Match statt, wird das Listenelement übernommen

Map

```
map BLOCK LIST
map EXPR,LIST
```

```
@squares = map (($_*$_), (1..20));
@squares = map {$_ , $_*$_} (1..20);
```

- mit map kann man aus einer Liste eine neue Liste erzeugen.
- Die neue Liste kann mehr Elemente enthalten als die ursprüngliche Liste

Reverse

```
@a = (7,8,9);
@b = reverse( @a );    # @b ist (9,8,7)
@b = reverse( 7, 8, 9 );     # wie oben

@b = reverse(@b);      # Liste umkehren
$hugo = reverse "Hugo";   # Hugo ist nun "oguH"
```

- Reverse kehrt die Reihenfolge einer Liste um
- im skalaren Kontext wird ein String in seiner Reihenfolge gedreht

Hochschule Darmstadt
Fachbereich Informatik

Systemprogrammierung mit Perl

4. Assoziative Arrays (Hashes)

Assoziative Arrays

- Kennzeichen eines assoziativen Arrays ("hash") ist die Paarung von jeweils zwei Elementen in der Form "Schlüssel-Wert".
- Ein Hash ist eine Ansammlung von Skalaren Daten, bei der die einzelnen Elemente über Indexwerte angesprochen werden können
- Die Indexwerte müssen keine kleinen positiven Integerzahlen sein, sondern sie können beliebige Skalare sein.
- Die Indexwerte werden auch als die Schlüssel des Hashes bezeichnet
- Gekennzeichnet wird ein assoziatives Array durch ein Prozentzeichen "%" vor dem Variablennamen.

Zugriff auf Hash Elemente

```
my %fred;
$fred {"aaa"} = "bbb";      # erzeugt den Schlüssel „aaa“ mit  Wert „bbb“
$fred {234.5} = 456.7;      # erzeugt den Schlüssel „234.5“ mit dem
                  #     Wert 456.7
print $fred{"aaa"};      # gibt „bbb“ aus
$fred{234.5} += 3;     # macht daraus den Wert 459.7
```

- Die Namensräume von Skalarvariablen, Arrayvariablen und Hashes sind vollkommen getrennt, so daß man einen Skalar $fred, eine Liste @fred und einen Hash %fred gleichzeitig benutzen kann (aber nicht sollte)
- Der Zugriff auf ein Element des Hashes %fred erfolgt über
  - ⇨ $fred{ $key }

Assoziative Arrays und Listen

```
%fred = (aaa => "bbb", 234.5 => 456.7);
                #%fred wird aus einer Liste  von
           # Literalen erzeugt
@fred_list = %fred;     # @fred_list wird zu ("aaa“, "bbb", "234.5“, 456.7)
%barney = @fred_list;       # barney wird wie %fred erzeugt
%barney = %fred;      # das gleiche wie vorher
```

- Es besteht die Möglichkeit Assoziative Arrays in Listen umzuwandeln und umgekehrt.
- Dabei wird jedes Schlüssel/Werte Paar als zwei hintereinanderliegende Elemente der Liste gespeichert
- Der Operator "=>" ist äquivqlent zum Komma Operator, nur muß der linke Operand nicht als String geschrieben werden, da ein automatisches Quoting erfolgt. Ausserdem erhöht der "=>" Operator die Lesbarkeit bei Schlüssel => Werte Paaren

Der Operator keys

```
%fred = (aaa => "bbb", 234.5 => 456.7);
@list = keys( %fred );  # @list wird zu ("aaa", 234.5) oder zu (234.5, "aaa")
foreach $key ( keys %fred ) {
      print „ bei $key steht $fred{$key} \n“;
}
```

```
if( keys (%irgendein_Array) ) { ... }
while (keys (%irgendein_Array) < 10 ) { ... }
```

- Mit dem Operator "keys %arrayname" holt man eine Liste aller aktuellen Schlüssel aus dem Hash %arrayname
- In einem skalaren Kontext gibt Keys die Anzahl der Elemente des Hashes zurück

Der Operator keys

```
my %squares = map {$_ => $_ * $_} (0..20);
for ( sort {$a <=> $b } keys %squares)
{
     print "$_ * $_  = $squares{$_}\n";
}
```

- Mit dem Operator map wird aus einer Liste ein Hash erzeugt
- mit keys werden die Schlüssel abgefragt und anschließend mit sort sortiert

Der Operator values

```
%lastname = ();
$lastname{ "Fred" } = "Feuerstein";
$lastname{ "barney" } = "Geroellheimer";
@lastname = values( %lastname );

# @lastname ist entweder („Feuerstein“, „Geroellheimer“) oder
#  umgekehrt
```

- Der Operator values(%arrayname) liefert eine Liste der aktuellen Werte von %arrayname

Der Operator each

```
while (($key,$value) = each %ENV)
{
    print "$key=$value\n";
}
```

- Der Operator each liefert ein Schlüssel/Wert Paar in einer Liste mit zwei Elementen
- Der Operator liefert mit jedem Zugriff auf das gleiche Array das nächstfolgende Schlüssel/Wert Paar, so lange bis alle Elemente gelesen wurden
- Wird dem ganzen Array ein neuer Wert zugewiesen, so wird each() auf den Anfang zurückgesetzt.
- Fügt man dem Array Elemente hinzu oder löscht welche heraus, so wird each() dadurch ziemlich verwirrt

Der Operator delete

```
%fred = (aaa => "bbb", 234.5 => 456.7);
delete $fred {aaa};
# %fred enthält jetzt nur noch ein Paar
```

- Mit dem Operator delete kann man Elemente aus einem assoziativen Array entfernen
- delete entfernt das Schlüssel/Wert Paar und gibt den Wert des gelöschten Paares zurück

Die Operatoren exists und defined

```
print "Exists\n"    if exists $hash{$key};
print "Defined\n"   if defined $hash{$key};
print "True\n"      if $hash{$key};

print "Exists\n"    if exists $array[$index];
print "Defined\n"   if defined $array[$index];
print "True\n"      if $array[$index];

print "Exists\n"    if exists &subroutine;
print "Defined\n"   if defined &subroutine;
```

- Mit exists wird die Existenz eines Elementes überprüft
- Mit defined wird überprüft ob das Element einen Wert ungleich undef besitzt, also initialisiert wurde
- Auch wenn eine Variable initialisiert wurde kann sie immer noch den Wert False besitzten (0 oder "")

Der Operator undef

```
undef $foo;
undef $bar{'blurfl'};        # Compare to: delete $bar{'blurfl'};
undef @ary;
undef %hash;
undef &mysub;
undef *xyz;                  # destroys $xyz, @xyz, %xyz, &xyz, etc.
```

- Der undef Operator macht die Definition einer skalaren Variablen, eines Arrays, eines Hashes oder eines Unterprogramms rückgängig
- Der undef Operator liefert als Rückgabewert immer den Wert undef
- Wird der undef Operator ohne folgenden Ausdruck aufgerufen so erhält man den Wert undef, dem man einer Variablen zuweisen kann, als Parameter übergeben oder als Returnwert in einem Unterprogramm nutzen kann

Hochschule Darmstadt
Fachbereich Informatik

Systemprogrammierung mit Perl

5. Referenzen und komplexe Datentypen

Referenzen

- Was ist eine Referenz?
  - ⇨ Eine Referenz ist ein Datum, das uns sagt wo der Ort eines anderen Datums ist
- Referenzen sind keine Pointer
  - ⇨ Eine Referenz zeigt immer auf einen definierten Ort für Daten. Welche Daten und welcher Typ von Daten sich dort befinden ist in den Daten gespeichert, nicht in der Referenz
- Referenzen sind Skalare
  - ⇨ Da Referenzen Skalare sind, können sie als Elemente in Arrrays und Hashes auftreten.
  - ⇨ Damit ist es möglich, alle komplexen Formen von Datenstrukturen zu beschreiben
- Anonyme Daten
  - ⇨ Mit Referenzen ist es möglich Daten zu erzeugen, denen keine direkte Variable zugeordnet ist, sondern die nur über eine Referenz angesprochen werden

Erzeugung von Referenzen

```
my @array = (1,2,3,4,5);
my $array_ref = \@array;

my %hash = (Fred => "Feuerstein", Barney => "Geröllheimer");
my $hash_ref = \%hash;

my $skalar = 42;
my $skalar_ref = \$skalar;

$coderef = \&handler;
$globref = \*foo;
```

- Es gibt zwei Wege eine Referenz zu erzeugen
- Wenn die Daten schon als Variable vorliegen, kann man den '\' Operator benutzen, um daraus eine Referenz zu erzeugen

Anonyme Referenzen

```
my $array_ref = [1,2,3,4,5];
my $hash_ref = {Fred => "Feuerstein", Barney => "Geröllheimer"};
my $coderef = sub {...}
```

- Um eine anonyme Referenz auf ein Array zu erzeugen, nutzt man eckige Klammeren '[...]' anstatt runder Klammern '(...)'
- Um eine anonyme Referenz auf einen Hash zu erzeugen, nutzt man geschweifte Klammern '{...}' anstatt runder Klammern '(...)'

Dereferenzierung

```
my @band = qw(Crosby Stills Nash Young);
my $ref  = \@band;
for (0..3) {
   print "Array    : ", $band[$_]  , "\n";
   print "Reference: ", ${$ref}[$_], "\n";
}
```

- Eine Dereferenzierung von Skalaren wird mit $, von Arrays mit @ und von Hashes mit % vorgenommen.

Dereferenzierung

```
my @array   = (1, 2, 3, 4, 5);
my $array_r = \@array;

print "This is our dereferenced array: @$array_r\n";
for (@$array_r) {
   print "An element: $_\n";
}
print "The highest element is number $#$array_r\n";
print "This is what our reference looks like: $array_r\n";
```

Arrow Operator

```
$arrayref = [1, 2, ['a', 'b', 'c']];
$arrayref->[2][1] ;      # liefert den Wert 'b'

$arrayref->[0] = "January"; # Array element
$hashref->{"KEY"} = "VALUE"; # Hash element
$coderef->(1,2,3); # Subroutine call
```

- Eine Dereferenzierung von Skalaren in Arrays oder Hashes oder der Zugriff auf Unterprogramme oder Objektmethoden kann auch über den Arrow Operator vorgenommen werden

Autovivication

```
$array[$x]->{"foo"}->[0] = "January";

$array[$x]{"foo"}[0]  = "January";

$score[$x][$y][$z] += 42;
```

- Perl legt automatisch einen anonymen Hash mit dem Schlüssel "foo" und ein anonymes Array mit dem Index 0 an
- Der Arrowoperator ist in diesem Fall optional, da Perl weiss, dass es sich nur um Referenzen handeln kann
- Auf diese Art können auch leicht mehrdimensionale Arrays beschrieben werden. Dabei besteht aber nicht die Forderung, dass alle Zeilen und spalten die gleiche Länge haben müssen

Beispiele

```
$bar = $$scalarref;
push(@$arrayref, $filename);
$$arrayref[0] = "January";
$$hashref{"KEY"} = "VALUE";
&$coderef(1,2,3);
print $globref "output\n";

$globref->print("output\n");     # iff IO::Handle is loaded

&{ $dispatch{$index} }(1,2,3);      # call correct routine
```

Beispiel zu Referenzen auf Funktionen

```
#!c:/perl/bin/perl.exe
my ( @Buchlist, $buchzeiger );
my ($anz);
my ($mittelwert);
my (%mfunc) = (
   1 => sub {$anz = @Buchlist; print " Die Anzahl der Buecher ist $anz \n" ; },
   2 => sub { &mittelwert; print "Der mittlere Preis betraegt $mittelpreis \n";},
   3 => \&teuer,
   4 => \&billig,
   5 => \&mittel,
   6 => \&nextBuch,
   7 => \&lastBuch,
   8 => \&searchBuch,
   9 => sub {exit();}
   );
```

- Hier wird ein Hash mit anonymen Unterprogrammen und Referenzen auf Unterprogramme gebildet

Beispiel zu Referenzen auf Funktionen

```
openfile();     # File einlesen
menu();         # print menue

while ( 1 ) {
    chomp ($prompt = <STDIN>);
    if ( ref $mfunc{$prompt} eq "CODE")
   {
      $mfunc{$prompt}() ;  # Arrow Operator zur Dereferenzierung ist hier optional
   }
   else
   {
       menu();
   }
}
```

Der Operator ref

```
if (ref($r) eq "HASH") {
      print "r is a reference to a hash.\n";
   }
unless (ref($r)) {
      print "r is not a reference at all.\n";
   }
```

- Der Operator ref liefert als Ergebnis true, ein String ungleich dem Nullstring, wenn der Ausdruck auf den er angewendet wurde eine Referenz ist.
- Ist der Ausdruck keine Referenz so ist das Ergebnis false
- Der Ergebnisstring ist "SCALAR", "ARRAY", "HASH", "CODE", "REF", "GLOB" oder "LVALUE", wenn es sich um einen dieser Typen handelt

Arrays of Arrays

```
@AoA = (
       [ "fred", "barney" ],
       [ "george", "jane", "elroy" ],
       [ "homer", "marge", "bart" ],
     );
```

Zugriff auf Arrays of Arrays

```
while ( <> ) {
    push @AoA, [ split ];                  # reading from file
}

for $i ( 1 .. 10 ) {
    $AoA[$i] = [ somefunc($i) ];                  # calling a function
}

push @{ $AoA[0] }, "wilma", "betty";          # add to an existing row

$AoA[0][0] = "Fred"; # one element

for $aref ( @AoA ) {                # print the whole thing with refs
    print "\t [ @$aref ],\n";
}
```

Hashes of Arrays

```
%HoA = (
       flintstones       => [ "fred", "barney" ],
       jetsons           => [ "george", "jane", "elroy" ],
       simpsons          => [ "homer", "marge", "bart" ],
     );
```

- Hashs von Arrays treten häufig bei XML Anwendungen auf

Erzeugung von Hashes of Arrays

```
# reading from file
# flintstones: fred barney wilma dino
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $HoA{$1} = [ split ];
}

# reading from file; more temps
# flintstones: fred barney wilma dino
while ( $line = <> ) {
    ($who, $rest) = split /:\s*/, $line, 2;
    @fields = split ' ', $rest;
    $HoA{$who} = [ @fields ];
}

# append new members to an existing family
 push @{ $HoA{"flintstones"} }, "wilma", "betty";
```

Arrays of Hashes

```
@AoH = (
      {
         Lead    => "fred",
         Friend  => "barney",
      },
      {
         Lead    => "george",
         Wife    => "jane",
         Son     => "elroy",
      },
      {
         Lead    => "homer",
         Wife    => "marge",
         Son     => "bart",
      }
 );
```

- Arrays von Hashes erhält man sehr häufig bei Datenbankzugriffen

Erzeugung von Arrays of Hashes

```
# reading from file
 # format: LEAD=fred FRIEND=barney
 while ( <> ) {
    $rec = {};
    for $field ( split ) {
       ($key, $value) = split /=/, $field;
       $rec->{$key} = $value;
    }
    push @AoH, $rec;
 }

 # reading from file
 # format: LEAD=fred FRIEND=barney
 # no temp
 while ( <> ) {
    push @AoH, { split /[\s+=]/ };
 }
```

Hashes of Hashes

```
%HoH = (
      flintstones => {
            lead      => "fred",
            pal       => "barney",
      },
      jetsons     => {
            lead      => "george",
            wife      => "jane",
            "his boy" => "elroy",
      },
      simpsons    => {
            lead      => "homer",
            wife      => "marge",
            kid       => "bart",
      },
 );
```

- XML Datenstrukturen werden üblicherweise auf Hashes of Hashes (HoH) und auf HoA's abgebildet

Records

```
$rec = {
        TEXT      => $string,
        SEQUENCE  => [ @old_values ],
        LOOKUP    => { %some_table },
        THATCODE  => \&some_function,
        THISCODE  => sub { $_[0] ** $_[1] },
        HANDLE    => \*STDOUT,
};
```

- Records oder Strukturen werden in Perl üblicherweise über Hashes abgebildet
- handelt es sich um einen Record so schreibt man die Schlüssel in Großbuchstaben um dies zu kennzeichnen

Ein Beispiel für komplexe Records

```
%TV = (
     flintstones => {
        series   => "flintstones",
        nights   => [ qw(monday thursday friday) ],
        members  => [
           { name => "fred",    role => "lead", age  => 36, },
           { name => "wilma",   role => "wife", age  => 31, },
           { name => "pebbles", role => "kid",  age  =>  4, },
        ],
     },
     jetsons     => {
        series   => "jetsons",
        nights   => [ qw(wednesday saturday) ],
        members  => [
           { name => "george",  role => "lead", age  => 41, },
           { name => "jane",    role => "wife", age  => 39, },
           { name => "elroy",   role => "kid",  age  =>  9, },
        ],
      },
    );
```

Hochschule Darmstadt
Fachbereich Informatik

Systemprogrammierung mit Perl

6. Packages, Module und Objekte

Modularisierung

- Um größere Programmpakete entwickeln zu können, und um Teile von Programmen wiederverwendbar zu gestalten ist es notwendig einen Mechanismus zur Modularisierung zu besitzen
- Die Einführung von Modulen gestattet es ein
  - ⇨ Programm auf verschiedene Files aufzuteilen,
  - ⇨ unterschiedliche Namensräume zu benutzen und
  - ⇨ die Schnittstellen zwischen den Modulen zu definieren

Definition von Packages

```
package Bankkonto;
our $saldo = 0;

sub einzahlen {
      my ( $betrag ) = @_;
      $saldo += $betrag;
      print „Sie haben jetzt $saldo Euro \n“;
}

sub abheben {
      my ( $betrag ) = @_;
      $saldo -= $betrag;
      $saldo = 0 if $saldo < 0;
      print „Sie haben jetzt $saldo Euro \n“;
}
```

- Das Schlüsselwort package bezeichnet den Anfang eines neuen Namensraums.
- Alle globalen Bezeichner die nach der package Anweisung eingeführt werden gehören zu dem neuen Namensraum
- Wird keine package Anweisung verwendet, nimmt Perl an, dass der aktuelle Packagename main ist. Alle Bezeichner gehören dann zum Package main.

Zugriff auf fremde Packages

```
package Geldautomat;
Bankkonto::einzahlen(10);
print $Bankkonto::saldo;
```

- Um auf einen Bezeichner in einem fremden Namensraum zuzugreifen, muß man den Package Namen vor den Variablennamen stellen.
- Wird kein Packagenamen angegeben, so sucht Perl den Namen im aktuellen Package

Packages und Dateien

- Die gleiche Package Deklaration kann in mehreren Dateien stehen.
- Ebenso können in einer Datei mehrere Package Deklarationen verwendet werden.
- Es ist empfehlenswert ein Package in einer Datei zu verwenden und der Datei den Namen „Packagename.pm“ zu geben, wobei Packagename der Name des jeweiligen Packages ist.
- Die Dateien können über die require oder die use Anweisung in das aktuelle Programm eingebunden werden.
- Die Use Anweisung wird bereits zur Compilezeit ausgeführt, die require Anweisung zur Laufzeit.
- Die use Anweisung ist daher vorzuziehen.

Initialisierung von Packages

```
sub BEGIN {
        print „ Modul xyz wurde geladen \n“;
}

sub END {
    print „Das war‘s \n“;
}
```

- Ist in einem Modul (Datei mit eigenem Package) eine Subroutine mit dem Namen BEGIN enthalten, so wird diese Funktion bereits zur Compilezeit ausgeführt.
- Enthält ein Modul einen END Block, so wird dieser ausgeführt, direkt bevor sich das Programm beendet. In diesem Programm können notwendige Aufräumarbeiten durchgeführt werden
- Ein INIT Block wird direkt vor der eigentlichen Ausführung in FIFO Ordnung gestartet,
- ein CHECK Block wird nach der vollständigen Compilierung und vor der Ausführung in LIFO Ordnung gestartet

Initialisierung von Packages

```
#!/usr/bin/perl
# begincheck

print                 " 8. Ordinary code runs at runtime.\n";

END { print          "14.   So this is the end of the tale.\n" }
INIT { print         " 5. INIT blocks run FIFO just before runtime.\n" }
CHECK { print       " 4.   So this is the fourth line.\n" }

print                 " 9.   It runs in order, of course.\n";

BEGIN { print       " 1. BEGIN blocks run FIFO during compilation.\n" }
END { print         "13.   Read perlmod for the rest of the story.\n" }
CHECK { print       " 3. CHECK blocks run LIFO at compilation's end.\n" }
INIT { print        " 6.   Run this again, using Perl's -c switch.\n" }

print                 "10.   This is anti-obfuscated code.\n";

END { print         "12. END blocks run LIFO at quitting time.\n" }
BEGIN { print       " 2.   So this line comes out second.\n" }
INIT { print        " 7.   You'll see the difference right away.\n" }

print                 "11.   It merely _looks_ like it should be confusing.\n";
__END__
```

Symbole exportieren und importieren

```
use Bankkonto(‘abheben‘, ‘einzahlen‘);
einzahlen(10);
```

```
package Bankkonto;
use Exporter;
our @ISA = (‘Exporter‘);          # Eigenschaften von Exporter erben
our @EXPORT_OK = (‘abheben‘, ‘einzahlen‘);

sub abheben { ...}
sub einzahlen {...}
```

- Manchmal wünscht man sich die Symbole eines Pakets ohne volle Qualifizierung zu benutzen.
- Das Modul muss dann die zu importierenden Namen exportieren
- Dazu stehen die Arrays @EXPORT_OK und @EXPORT zur Verfügung
- Variablen in @EXPORT_OK müssen explizit in der use Anweisung importiert werden,
- Variablen in @EXPORT werden automatisch in den Namensraum des aufrufenden importiert

Symboltabellen

- Jedes Package besitzt seine eigene Symboltabelle, die als Hash realisiert ist (auch stash = Symbol Table Hash genannt).
- Diese stashes stehen direkt zur Verfügung und können vom Benutzer abgefragt werden
  - ⇨ Die Symboltabelle von main ist unter %main:: zugreifbar,
  - ⇨ die Symboltabelle des Pakets Foo kann über %Foo:: erreicht werden.
- Jeder Eintrag zeigt auf einen Typeglob der wiederum auf einen oder mehrere Werte zeigt. Mögliche Werte sind:
  - ⇨ Skalar, Array, Hash, Subroutine, Dateihandle, Formatname, Verzeichnishandle

Objekte in PERL

- Ein Objekt ist ein referenziertes Dingsda, das weiß zu welcher Klasse es gehört
- Eine Klasse ist ein Paket, welches Methoden bereitstellt, die mit Objekten arbeiten
- Eine Methode ist eine Unterroutine, die eine Objektreferenz (oder bei Klassenmethoden einen Paketnamen) als ersten Parameter erwartet

Objekte als referenziertes Dingsda

- Ein Konstruktor ist eine Unterroutine, die eine Referenz auf ein Dingsda zurückgibt, das einer Klasse zugeordnet ist
- Der Konstruktor verwendet dazu die Funktion bless, die ein Objekt zu einer Klasse zugehörig markiert.
- Die Funktion bless erwartet ein oder zwei Argumente. Das erste Argument ist immer die Objektreferenz, die mit dem Namen einer Klasse „gesegnet“ wird.
- Fehlt das zweite Argument, so wird der Name des aktuellen Pakets verwendet, ansonsten enthält das zweite Argument den Namen des Pakets, zu dem das Objekt gehören soll.

Objekte „segnen“

```
package dingsda;        # der einfachste Konstruktor
sub new {
      return bless {};
}
```

```
package dingsda;        # ausführlicher aber mit gleicher Funktionalität
sub new {
      my $self = {};    # Referenz auf Hash erzeugen
      bless $self;      # Referenz segnen
      return $self;     # Referenz zurückgeben
}
```

```
package dingsda;        # erweiterte Funktionalität
sub new {
      my $class = shift;        # Name der Klasse ist erstes Argument
      my $self = {};            # eine Referenz erzeugen
      bless $self, $class;      # das Objekt segnen
      $self->_initialize();     # das Objekt initialisieren
      return $self;             # die Referenz zurückgeben
}
```

Klassen

- Eine Klasse ist einfach ein Paket
- Alle exportierten Funktionen können als Methoden benutzt werden und erhalten als ersten Parameter eine Referenz auf das Objekt.
- Innerhalb eines Paketes teilt das Array @ISA mit, wo noch nach Methoden zu suchen ist. Die Pakete werden rekursiv von unten anfangend nach Methoden durchsucht.
- Durch das @ISA Array ist in Perl sowohl einfache als auch multiple Vererbung möglich

Klassen- und Instanzmethoden

- Eine Klassenmethode erhält als erstes Argument den Klassennamen übergeben.
- Der Klassenname ist ein String und keine Referenz
- Die wichtigste Anwendung für Klassenmethoden ist die Erzeugung von Objekten einer Klasse
- Instanzmethoden erhalten als erstes Argument die Referenz auf das Objekt.
- Üblicherweise wird diese Referenz mit shift einer internen Variablen zugewiesen (meist self oder this genannt, je nach Geschmack des Programmierers)

Aufruf von Methoden

```
# Beispiel: Indirekter Methodenaufruf
$mw = new MainWindow;
```

```
# Beispiel: direkter Methodenaufruf
my $mw = MainWindow->new();
$mw->title(„Hallo Welt“);
$mw->Button( -text => „Fertig“, -command => sub {exit } )->pack;
```

Perl stellt zwei Formen des Methodenaufrufs zur Verfügung

- 1. Indirekter Methodenaufruf

  METHODE KLASSE_ODER_INSTANZ LISTE

  ⇨ Der indirekte Methodenaufruf kann zu Schwierigkeiten bei komplizierteren Konstruktionen führen

- 2. Objektorientierte Syntax

  KLASSE_ODER_INSTANZ->METHODE (LISTE)

Destruktoren

- Wenn die letzte Referenz auf ein Objekt verschwindet, wird das Objekt automatisch durch die Garbage Collection entfernt.
- Durch die Definition einer DESTROY Methode kann man erreichen, dass diese direkt vor der Vernichtung des Objekts aufgerufen wird

Instanzvariablen

```
package Foo;
sub new {
      my $type = shift;
      my %params = @_;
      my $self = {};
      $self->{'High'} = $params{'High'};
      $self->{'Low'}  = $params{'Low'};
      bless $self, $type;
}

package main;
$a = Foo->new( 'High' => 42, 'Low' => 11 );
print "High=$a->{'High'}\n";
print "Low=$a->{'Low'}\n";
```

- Instanzvariablen werden üblicherweise durch einen Eintrag in einen Hash realisiert

Instanzvariablen und Vererbung

```
package Bar;
sub new {
	my $type = shift;
	my $self = {};
	$self->{'buz'} = 42;
	bless $self, $type;
}

package Foo;
our @ISA = qw( Bar );
sub new {
	my $type = shift;
	my $self = Bar->new;
	$self->{'biz'} = 11;
	bless $self, $type;
}

package main;
$a = Foo->new;
print "buz = ", $a->{'buz'}, "\n";
print "biz = ", $a->{'biz'}, "\n";
```

- Abgeleitete Klassen können jederzeit das Objekt der Vaterklasse erweitern und dann neu "segnen"
- Die Vererbung wird durch einen Eintrag im Array @ISA erzeugt

Containing and Using

```
package Bar;
sub new {
my $type = shift;
my $self = {};
$self->{'buz'} = 42;
bless $self, $type;
}

package Foo;
sub new {
my $type = shift;
my $self = {};
$self->{'Bar'} = Bar->new;
$self->{'biz'} = 11;
bless $self, $type;
}

package main;
$a = Foo->new;
print "buz = ", $a->{'Bar'}->{'buz'}, "\n";
print "biz = ", $a->{'biz'}, "\n";
```

- In Perl werden die Beziehungen
  - ⇨ Assoziation,
  - ⇨ Aggregation und
  - ⇨ Komposition
- über Referenzen und komplexe Datenstrukturen realisiert

Hochschule Darmstadt
Fachbereich Informatik

Systemprogrammierung mit Perl

7. Kontrollstrukturen in Perl

Kontrollstrukturen: wahr oder falsch

```
0     # wird falsch "0"
1-1   # wird falsch "0"
1     # wird wahr "1"
""    # leere Zeichenkette, also falsch
"1"   # nicht "" oder "0" also wahr
"00"      # mehr als ein Zeichen "0", also wahr
"0.00"    # wahr
undef     # wird zu "", also falsch
```

- Was ist wahr oder falsch?
  - ⇨ Der Kontrollausdruck wird als Zeichenkette betrachtet.
  - ⇨ Wenn die Zeichenkette die Länge 0 hat ("") oder nur aus dem Zeichen "0" besteht ist der Ausdruck falsch, ansonsten wahr

if, elsif, else

```
iif ($weather eq "snowing") {
      print "OK, let's go!\n";
} elsif ($weather eq "raining") {
      print "No way, sorry, I'm staying in.\n";
} elsif ($temperature < 18) {
      print "Too cold for me!\n";
} elsif ($work > 30) {
      print "Sorry - just too busy.\n";
} else {
      print "Well, why not?\n";
}
```

- Im Gegensatz zu C++ oder Java muss in Perl hinter einem if immer ein Block folgen
  - ⇨ dies ist deutlich sicherer
  - ⇨ Die Klammern um die logischen Ausdrücke sind optional, da Perl weiß, dass nach dem Ausdruck ein Block folgen muss.
- um trotzdem eine Reihe von Fällen überprüfen zu könne gibt es die Anweisung *elsif (Ausdruck) Block*, die dies gewährleistet und die immer hinter einer *if* Anweisung stehen muss

unless

```
unless ( defined $_[0] )
{
     die "Der Parameter 0 ist nicht definiert";
}
```

- Die unless Anweisung ist eine negierte if Anweisung und wird häufig zur Fehlerüberprüfung eingesetzt

while / until

```
my $countdown = 5;

while ($countdown-- > 0) {
      print "Counting down: $countdown\n";
}
```

```
my $countdown = 5;

until ($countdown-- == 0) {
      print "Counting down: $countdown\n";
}
```

- while und until sind Schleifenanweisungen in Perl
  - ⇨ Der Block nach while wird ausgeführt wenn der Ausdruck hinter while true ist
  - ⇨ Der Block nach until wird ausgeführt, wenn der Ausdruck hinter until false ist
  - ⇨ Die Klammern um die logischen Ausdrücke sind optional, da Perl weiß, dass nach dem Ausdruck ein Block folgen muss.

for Schleife

```
for( $i = 1; $i <= 10; $i++)     #arithmetische Form
{
    print "$i  \n";
}
```

```
for $i (1..10)              # Iterator Form
{
    print "$i  \n";
}
```

- Neben der for oder foreach Schleife die über ein Array iteriert gibt es in Perl wie in C++ oder Java die arithmetische for loop mit Startausdruck, Bedingung und Increment Ausdruck
- Die die arithmetische for loop wird deutlich seltener benutzt als die Iterator Form
- Die Skalarvariable der Iterator Schleife ist nur lokal in der Schleife gültig

Beispiele zur Iterator For Schleife

```
@a = (1, 2, 3, 4, 5);
foreach $b ( reverse @a ) {
      print $b;
}

@a = (1, 2, 3, 4, 5);
foreach ( reverse @a ) {
      print ;
}
```

- wird in der foreach Anweisung keine Skalarvariable angegeben, so wird der Wert der Variablen $_ zugewiesen
- Ein print Ausdruck ohne Liste druckt die Defaultvariable $_ aus

weitere Kontrollstrukturen: last

```
  LINE: while (<STDIN>) {
      last LINE if /^$/;    # exit when done with header
      #...
    }
# last springt hierher
```

- mit last wird die innerste Schleife verlassen
- mit last und einem folgenden Label wird die Schleife mit dem entsprechenden Label verlassen
- der Operator last entspricht dem break in C, C++ und Java

weitere Kontrollstrukturen: next

```
LINE: while (<STDIN>) {
    next LINE if /^#/;  # discard comments
    #...
    ...
    # next springt hierher
  }
```

- der Operator next startet den nächsten Schleifendurchgang
- der Operator next entspricht dem continue in C, C++ und Java

weitere Kontrollstrukturen: redo

```
LINE: while (defined($line = <ARGV>)) {
      chomp($line);
      if ($line =~ s/\\$//) {
         $line .= <ARGV>;
         redo LINE unless eof(); # not eof(ARGV)!
      }
      # now process $line
}
```

- Redo startet den Schleifendurchlauf erneut, ohne dass die Bedingung noch einmal überprüft wird

weitere Kontrollstrukturen: Schleifen mit Labels

```
OUTER: for (my $i = 1; $i <= 10; $i++) {
     INNER: for (my $j = 1; $j <= 10; $j++) {
          if ( $i*$j == 63 ) {
               print “$i mal $j ist 63!\n“;
               last OUTER;
          }
          if ( $j >= $i ) {
               next OUTER;
          }
     }
}
```

- Durch die Nutzung von Labels kann gezielt eine geschachtelte Schleifenstruktur durch last, next und redo kontrolliert werden

Modifikatoren

```
print "Basset hounds got long ears" if length $ear >= 10;
go_outside() and play() unless $is_raining;

print "Hello $_!\n" foreach qw(world Dolly nurse);

# Both of these count from 0 to 10.
print $i++ while $i <= 10;
print $j++ until $j >  10;
```

- einfache Statements können um einen Modifikator erweitert werden
- Modifikatoren sind die Kontrollstrukturen
  - ⇨ if EXPR
  - ⇨ unless EXPR
  - ⇨ while EXPR
  - ⇨ until EXPR
  - ⇨ foreach LIST

Switch Statements in Perl

```
use Switch;

switch ($val) {

      case 1          { print "number 1" }
      case "a"  { print "string a" }
      case [1..10,42]     { print "number in list" }
      case (@array)       { print "number in list" }
      case /\w+/      { print "pattern" }
      case qr/\w+/  { print "pattern" }
      case (%hash) { print "entry in hash" }
      case (\%hash)       { print "entry in hash" }
      case (\&sub)  { print "arg to subroutine" }
      else        { print "previous case not true" }
}
```

- Perl kennt kein eigenes Switch Statement, aber ab Perl 5.8 gibt es ein Modul Switch.pm das einen flexiblen Switch zur Verfügung stellt
- Weitere Informationen in der Moduldokumentation

Der Rhombus Operator (Diamond Operator)

```
#!/usr/bin/perl
while ( <> ) {
      print $_;
}
```

- Der Rhombus Operator bekommt seine Daten von Dateien, deren Namen in der Kommandozeile angegeben werden können
- Gibt man auf der Kommandozeile keinen Dateinamen an, so liest der Rhombus Operator automatisch von der Standardeingabe
- Aufruf des Beispiels mit der Liste Datei1 Datei2 Datei3 druckt alle Zeilen der Dateien Datei,Datei2 und Datei3 aus

Ausgabe auf STDOUT

```
print (2+3), "hallo";
print ((2+3), "hallo");
print 2+3, "hallo";
printf "%15s %5d $10.2f\n", $s, $n , $r;
```

- print: dem Operator print wird eine Liste von Zeichenketten übergeben, die er der Reihe nach auf die Standardausgabe schickt
- printf: formatierte Ausgabe. Printf erhält eine Liste von Argumenten, deren erstes die Formatierungsanweisung ist

Hochschule Darmstadt
Fachbereich Informatik

Systemprogrammierung mit Perl

8. Reguläre Ausdrücke

Reguläre Ausdrücke

```
while (<>) {
      if ( /abc/ ) {
            print “$_“;
      }
}
```

- In Perl wird eine Zeichenkette als „Regulärer Ausdruck“ interpretiert, wenn sie in Schrägstriche eingeschlossen ist
- Eine alternative Schreibweise ist m[...] oder m#...# oder m%...%. Als Start- und Endezeichen können die diversen Klammern und andere nicht alphanumerische Zeichen genutzt werden
- Der Mustervergleich findet, wenn nichts anderes angegeben ist, mit der Variablen $_ statt und liefert die Anzahl der Treffer zurück

Substitute Operator

```
while (<>) {
      s/ab*c/def/;
      print “$_“;
}
```

- Der Substitute Operator ermöglicht es einen Teil einer Zeichenkette, der mit dem regulären Ausdruck übereinstimmt, durch eine andere Zeichenkette zu ersetzen.

Muster für Einzelzeichen und Zeichenklassen

```
[0123456789]     # jede Ziffer
[0-9]            # das gleiche
[0-9\-]          # 0-9 und Minus
[a-z0-9]         # Kleinbuchstabe und Ziffern
[A-Za-z0-9_]  # Buchstabe Ziffer oder Underscore
[0-9\-+.eE]   # Zeichen die in einer Zahl vorkommen können
```

- Ein Zeichen „a“ im RE erwartet auch ein Zeichen „a“ in der Zeichenkette
- Ein „.“ im RE matched jedes Zeichen außer Newline (\n)
- Eine Zeichenklasse wird durch eckige Klammern im RE definiert.
- [abcdef] matched jedes der Zeichen in der Zeichenkette, aber es erfolgt insgesamt nur ein match

Negierte Zeichenklassen

```
[^0-9]          # keine Ziffer
[^aeiouAEIOU]    # jedes Zeichen, das kein Vokal ist
[^\^]           # alle Zeichen außer „^“
```

- wird als erstes Zeichen einer Zeichenklasse ein „^“ gewählt, so handelt es sich um eine negierte Zeichenklasse. Es werden alle Zeichen außer den angegebenen gematched

vordefinierte Zeichenklassen

Konstrukt	Klasse	verneintes Konstrukt	verneinte Zeichenklasse
\d (digits)	[0-9]	\D (digits, not!)	[^0-9]
\w (words)	[a-zA-Z0-9_]	\W (words, not!)	[^a-zA-Z0-9_]
\s (space)	[\r\t\n\f]	\S (space, not!)	[^ \r\t\n\f]

Zeichenfolgen und Vervielfachung

```
/a.{5}b/       #sucht ein Muster, das mit a beginnt, fünf beliebige Zeichen
          # ausser Newline besitzt und mit b aufhört
```

- Eine Zeichenfolge besteht aus einer Reihe von Einzelzeichen und wird direkt gematched
- Ein „*“ steht für null oder mehr Zeichen des unmittelbar vorangehenden Zeichens oder der Zeichenklasse
- ein „+“ steht für ein oder mehr Zeichen
- ein „?“ steht für null oder ein Zeichen
- ein {5,10} steht für 5 bis 10 Zeichen (allgemeiner Multiplikator)
- ein {5,} steht für mindestens 5 Zeichen
- ein {5} steht für genau 5 Zeichen

Gierige Operatoren

```
# Beispiel: Suche nach dem kleinsten C-Block
/\{.*?\}/
```

- Die Operatoren * und + versuchen jeweils möglichst viele Zeichen zu matchen. Es wird also bei Einsatz dieser Operatoren das größtmögliche Muster erkannt.
- Man kann diesen Operatoren die Gier abgewöhnen und das kleinste Muster suchen, indem man hinter diese Operatoren ein Fragezeichen stellt

Zwischenspeicher

```
/a(.*)b\1c/;    # matched zum Beispiel aFREDbFREDc

$_ = “a xxx b yyy c zzz d“;
s/b(.*)c/d\1e/;      # wie sieht der String danach aus?

(?:a|b)         # entweder a oder b aber Klammer wird nicht mitgezählt
```

- wird ein Teil des Musters durch runde Klammern gekennzeichnet, so wird der gematchte Teil gespeichert und kann anschließend wiederverwendet werden
- Als Zwischenspeicher stehen die Variablen \1 bis \9 zur Verfügung
- Sie werden nacheinander jeweils mit den geklammerten Ausdrücken der Suchstrings besetzt
- besonders wichtig sind die Zwischenspeicher bei Ersetzungsoperationen
- Wenn Klammern benutzt, so werden sie automatisch zwischengespeichert und mitgezählt. Will man das verhindern so kann als erstes Zeichen hinter der öffnenden Klammer ein ?: verwenden

Spezialvariablen

```
$_ = "nur ein kleiner Test";
/kl.*er/;   # Uebereinstimmung mit kleiner
# $` ist jetzt „nur ein “
# $& ist jetzt „kleiner“
# $´ ist jetzt „ Test“
```

- $& Teil der Zeichenkette der übereinstimmt
- $` Teil der Zeichenkette vor dem Ausdruck
- $´ Teil der Zeichenkette nach dem Ausdruck
- $1..$9 Zwischenspeicher, die nach dem Matching auf \1..\9 gesetzt werden

Alternativen

- Alternativen werden durch einen „|“ gekennzeichnet.
- /hugo|otto/ matched entweder den String hugo oder otto

Verankerung von Zeichenketten

```
/Fred\b/;       # matched Fred aber nicht Freddy
/\bwiz/;        # matched wizard aber nicht qwiz
/\b+\b/;        # matched „x+y“ aber nicht „x++y“ oder „x + y“
/abc\bdef/;     # kein match möglich
/^\#.*/         # alle Zeilen die mit einem Hash beginnen
/^\s+\#.*/      # alle Zeilen in denen vor dem Hash nur Spaces sind
```

- Zeichenmuster können an speziellen Stellen verankert werden:
- \b gibt an, dass an dieser Stelle eine Wortgrenze sein muß
- \B gibt an, dass an dieser Stelle keine Wortgrenze sein darf
- ^ Das Zeichen Caret bedeutet der Zeilenanfang (wenn es an erster Stelle im Muster steht)
- $ Das Dollarzeichen kennzeichnet das Zeilenende

Rangfolge von Operatoren

Name	Darstellung
Klammern	()
Multiplikatoren	+ * ? {m,n}
Zeichenfolgen und Anker	abc ^ $ \b \B
Alternativen	|

Der Matching Operator

```
$a = “Hallo Welt“;
$a =~ /^Ha/;              # wahr
$a =~ /(.)\1/;       # wahr
if ( $a =~ /(.)\1/ ) {     # wahr und weiter
     irgendein Ausdruck;
}
```

- Der Operator =~ ist der Matching Operator
- Befindet sich die zu untersuchende Zeichenkette nicht in der Variablen $_, so kann man mit dem Matching Operator einen Mustervergleich vornehmen
- Der Operator !~ ist der negierte Matching Operator
- Der matching Operator kann auf alles angewandt werden, was einen skalaren String liefert

Beispiel für den Matching Operator

```
print “Noch irgendwelche Wünsche? “;
if ( <STDIN> =~ /^[jJ]/ ) {
      print “Und um welchen Wunsch koennte es sich handeln? “;
      <STDIN>; # naechste Eingabe wegschmeissen
      print “Tut mir leid, aber das kann ich nicht machen.\n“;
}
```

- Einen skalaren Stringwert liefert auch <STDIN>

Modifikatoren

Modifier	Bedeutung
g	Globale Suche, d.h. finde alle Vorkommen
i	ignoriere Groß/Kleinschreibung
m	Behandle String, als würde er aus mehreren Zeilen bestehen
o	kompiliere Muster nur einmal
s	Behandle String wie eine Zeile
x	Verwende erweiterte reguläre Ausdrücke

- Es kommt vor, daß man eine Zeichenkette vergleichen möchte, aber Groß- und Kleinschreibung zulassen möchte
  - ⇨ Ein an das Muster angehängter Modifier (i) kann dieses Verhalten (ignore case) erreichen
- Manchmal will man alle Treffer eines Mustervergleichs erhalten
  - ⇨ der Modifikator g ermöglicht dies

Beispiele

```
# ignorieren der Groß und Kleinschreibung
$muster = “buffalo“;
if ( /$muster/i ) {

# Listenkontext -- extrahiere drei numerische Felder
($one, $five, $fifteen) = (`uptime` =~ /(\d+.\d+)/g );
```

```
#Skalarer Kontext -- Berechne die Anzahl der Sätze
$/ = "";   # Absatzmode
while ( $paragraph = <> ) {
      while ( $paragraph =~ /[a-z][‘“)]*[.!?]+[‘“)]*\s/g ) {
            $sentences++;
      }
}
print "$sentences\n";
```

Beispiel für erweiterte reguläre Ausdrücke

```
# finde doppelte Wörter in Absätzen, die sich auch über Zeilengrenzen
# hinweg erstrecken können. Verwende /x für Leerzeichen und
# Kommentare
$/ = "";  # Absatzmode
while (<>) {
      while ( m{
                \b          # beginne an einer Wortgrenze
                (\w\S+)         # finde einen Wortteil,
                (
                     \s+  # getrennt durch irgendein Whitespace
                     \1   # wieder dasselbe Wort
                )+           # wiederholen
                \b           # bis zur nächsten Wortgrenze
              }xig )
      {
           print “Doppeltes Wort $1 in Absatz $. \n“;
      }
}
```

Variablenersetzung

```
$sentence = “Jeder richtige Vogel kann fliegen“;
print “Wonach soll ich suchen? “;
$what = <STDIN>;
chomp $what;

if ( $sentence =~ /\b$what\b/ ) {
      print “Der Satz enthält das Wort  $what!\n“;
}

# sichere Variante mit einfacher Ersetzung
if ( $sentence =~ /\b\Q$what\E\b/ ) {
      print “Der Satz enthält das Wort  $what!\n“;
}
```

- In einem regulären Ausdruck findet eine Variablenersetzung statt.
- Schließt man die Variable in \Q und \E ein, so wird der Wert der Variablen wörtlich genommen und nicht weiter interpretiert

Hochschule Darmstadt
Fachbereich Informatik

Systemprogrammierung mit Perl

9. Funktionen und Unterprogramme

Funktionen

- Aufruf von Funktionen
  - ⇨ Funktionen werden durch ihren Namen aufgerufen:

  hello; oder hello();

  - ⇨ oder durch den Namen mit einem vorangestellten &:

  &hello;
- Rückgabewerte
  - ⇨ Der Rückgabewert einer Funktion ist der Wert des letzten berechneten Ausdrucks im Rumpf des Unterprogramms

```
sub sum_of_a_and_b {
    $a + $b;
}
```

Funktionen und Listen

```
sub list_of_a_and_b {
     ($a, $b);
}

$a = 5; $b = 6;
@c = &list_of_a_and_b;      # $c enthält den Wert (5,6)
```

- Funktionen können auch Listen zurückgeben
- Bei großen Listen sollten man aber mit Referenzen arbeiten

Parameter

```
#    Zugriffsmechanismen auf Parameter in Unterprogrammen

print „dies ist der erste Parameter: $_[0] \n“;

$anz = @_;
print „die Anzahl der Parameter: $anz \n“;

($a, $b, @c) = @_;      # die ersten beiden Parameter nach $a und $
                        # den Rest nach @c
$a = shift @_;          # den ersten Parameter nach $a und aus @_
```

- Fügt man beim Aufruf eines Unterprogramms hinter den Namen eine Liste, so wird diese Liste automatisch der Variablen @_ zugewiesen und besteht während der Laufzeit des Unterprogramms
- Das Unterprogramm kann diese Variable lesen und Werte und Anzahl der Argumente feststellen

local

- Will man keine rein lokale Variablen für ein Unterprogramm erzeugen, so kann man den Operator local statt dem Operator my benutzen
- Die mit local erzeugten Variablen sind in allen auf-gerufenen Unterprogrammen bekannt!!
- Local ist weit weniger effizient als my

Hochschule Darmstadt
Fachbereich Informatik

Systemprogrammierung mit Perl

10. Zugriffe aufs Filesystem

open und Filehandles

```
open(INFO,      "datafile")         or   die("can't open datafile: $!");
open(INFO,    "<  datafile")        or   die("can't open datafile: $!");
open(RESULTS,">  runstats")         or   die("can't open runstats: $!");
open(LOG,     ">> logfile ")        or   die("can't open logfile:  $!");
# Filehandle schließen
close(LOG);
```

- Perl stellt zu Programmstart drei Filehandles zur Verfügung: STDIN, STDOUT, STDERR
- Mit dem Operator open können weitere Filehandles geöffnet werden
- Mit dem Operator close können Files wieder geschlossen werden.
- Bei Programmende werden automatisch alle Files geschlossen

weitere Formen von open

- # open FILEHANDLE,EXPR
  - ⇨ Zwei Parameter Form von Open.
  - ⇨ Die Angabe des Modes wird implizit im String mitgegeben
- # open FILEHANDLE,MODE,EXPR
  - ⇨ Drei Parameter Form von Open
  - ⇨ sicherste Form. Sollte bevorzugt werden
  - ⇨ Mode wird explizit als 2. Parameter übergeben
- # open FILEHANDLE,MODE,EXPR,LIST
  - ⇨ Die Liste der übrigen Parameter wird dem Ausdruck als Parameter übergeben, falls es sich bei Ausdruck um ein Kommado handelt
- # open FILEHANDLE,MODE,REFERENCE
  - ⇨ Es besteht die Möglichkeit ein Filehandle dirket zum Memory zu öffen
- # open FILEHANDLE
  - ⇨ Sollte man nicht benutzen

Modes für open

- '<' öffnen zum Lesen
- '+<' öffnen zum Lesen und Schreiben
- '>' öffen zum Schreiben, File wird vorher geleert
- '+>' öffen zum Schreiben und Lesen, File wird vorher geleert
- '>>' öffnen zum Anhängen
- '+>>' öffnen zum Lesen und Anhängen
- '-|' öffnen einer Input Pipe
- '|-' öffnen einer Output Pipe
- "<:utf8" öffnet File zum lesen und schaltet gleichzeitig das UTF8 Filter ein (siehe Modul PerlIO)

Open und Pipes

```
open(PRINTER, "| lpr -Plp1")    || die "can't run lpr: $!";
print PRINTER "stuff\n";
close(PRINTER)                  || die "can't close lpr: $!";

open(NET, "netstat -i -n |")    || die "can't fork netstat: $!";
while (<NET>) { }               # do something with input
close(NET)                      || die "can't close netstat: $!";
```

- mit open können Pipes geöffnet und die entsprechenden Prozesse gestartet werden
- Das Pipe Symbol '|' wird in der 2 Parameter Form dem Kommando für Input Pipes vorangestellt, für Output Pipes hintangestellt
- In der 3 Parameterform wird die Richtung der Pipe durch '-|' oder '|-' ausgedrückt

Fehlerbehandlung beim öffnen von Files

- Die Fehlerbehandlung erfolgt üblicherweise mit dem Operator die der über ein or an die openfunktion geküpft wird
- der aktuelle Fehlertext ist in der Variablen $! enthalten und kann im Text von di benutzt werden
- Wird „die“ ohne abschließendes Newline (\n) benutzt, so gibt Perl noch Dateinamen und Zeilennummer aus

Beispiele zu open

```
$ARTICLE = 100;
  open ARTICLE or die "Can't find article $ARTICLE: $!\n";
  while (<ARTICLE>) {...

  open(LOG, '>>/usr/spool/news/twitlog');    # (log is reserved)
  # if the open fails, output is discarded

  open(DBASE, '+<', 'dbase.mine')              # open for update
    or die "Can't open 'dbase.mine' for update: $!";

  open(DBASE, '+<dbase.mine')                                  # ditto
    or die "Can't open 'dbase.mine' for update: $!";

  open(ARTICLE, '-|', "caesar <$article")     # decrypt article
    or die "Can't start caesar: $!";

  open(ARTICLE, "caesar <$article |")                          # ditto
    or die "Can't start caesar: $!";
```

Beispiele zu open

```
open(EXTRACT, "|sort >Tmp$$")              # $$ is our process id
     or die "Can't start sort: $!";

   # in memory files
open(MEMORY,'>', \$var)
     or die "Can't open memory file: $!";
print MEMORY "foo!\n";                     # output will end up in $var
```

komplizierteres Beispiel

```
# process argument list of files along with any includes

foreach $file (@ARGV) {
   process($file, 'fh00');
}

sub process {
   my($filename, $input) = @_;
   $input++;              # this is a string increment
   unless (open($input, $filename)) {
      print STDERR "Can't open $filename: $!\n";
      return;
   }

   local $_;
   while (<$input>) {        # note use of indirection
      if (/^#include "(.*)"/) {
         process($1, $input);
         next;
      }
      #...           # whatever
   }
}
```

Arbeiten mit Filehandles

```
# Lesen aus einem File
open(EP, “/etc/passwd“);
while( <EP> ) {
      chomp;
      print „$_ ist in der Passwortdatei enthalten!\n“;
}
# Schreiben in ein File
print LOGFILE “$n wurde fertiggestellt \n“;
print STDOUT “Hello world \n“;
```

- Das Lesen aus einem File erfolgt üblicherweise durch "<FILEHANDLE >". Dabei wird im Standard eine Zeile gelesen. Andere Leseformen können über den INPUT_RECORD_SEPARATOR $/ eingestellt werden
  - ⇨ $/ = ""; # lesen bis Leerzeile
  - ⇨ $/ = undef; # lesen der ganzen Datei
- Beim Schreiben ist zu beachten, dass zwischen Filehandle und der Liste kein Komma steht

Update von Files

```
open(FH, "+<", $FILE)          or die "Opening: $!";
@ARRAY = <FH>;
# change ARRAY here
seek(FH,0,0)                   or die "Seeking: $!";
print FH @ARRAY                or die "Printing: $!";
truncate(FH,tell(FH))          or die "Truncating: $!";
close(FH)                      or die "Closing: $!";
```

- Das File wird zum update geöffnet ("+<");
- Das File wird vollständig in den Speicher gelesen, da ein update im File nur bei einer festen Satzgröße sinnvoll ist
- mit seek wird der Filezeiger wieder auf den Anfang gesetzt
- mit truncate wird dem file wieder die richtige Länge gegeben. tell liefert uns dabei die Längeninformation

Datei Tests

```
# Test auf Existenz
if(-e $a ) {
      die „Datei $a existiert schon\n“;
}

# Test auf Lesbarkeit
foreach ( @some_list_of_filenames) {
      print “$_ ist lesbar\n“ if -r;
}
```

- Mit einem Dateitest kann man überprüfen, ob eine Datei vorhanden ist, ob sie lesbar ist ...

Datei Tests

Datei Test	Bedeutung
-r	Datei oder Verzeichnis ist lesbar
-w	Datei oder Verzeichnis ist schreibbar
-x	Datei oder Verzeichnis ist ausführbar
-o	Datei oder Verzeichnis gehört dem Benutzer
-e	Datei oder Verzeichnis existiert
-z	Datei existiert und hat die Größe Null
-s	Datei oder Verzeichnis existiert und hat eine Größe ungleich Null
-f	Argument ist eine normale Datei
-d	Argument ist ein Directory
-l	Argument ist ein symbolischer Link
-S	Argument ist ein Socket
-t	Datei ist ein Ausgabegerät
-T	Datei enthält Text
-M	Zeit seit der letzten Änderung in Tagen
-A	Zeit seit dem letzten Zugriff in Tagen
-C	Zeit seit der letzten Änderung des INODE in Tagen

```
($dev, $ino, $mode, $nlink, $uid, $gid, $rdev, $size, $atime, $mtime, $ctime, $blksize,
$blocks) = stat (...);
($uid, $gid) = stat("/etc/passwd")[4,5];
```

returnwerte von stat

stat		
0	dev	device number of filesystem
1	ino	inode number
2	mode	file mode (type and permissions)
3	nlink	number of (hard) links to the file
4	uid	numeric user ID of file's owner
5	gid	numeric group ID of file's owner
6	rdev	the device identifier (special files only)
7	size	total size of file, in bytes
8	atime	last access time in seconds since the epoch
9	mtime	last modify time in seconds since the epoch
10	ctime	inode change time in seconds since the epoch (*)
11	blksz	preferred block size for file system I/O
12	blcks	actual number of blocks allocated

chdir und globbing

```
chdir (“/etc“) or die “Wechsel nach /etc nicht möglich“; # Wechsel von Verzeichnis
@a = </etc/host*>;   # Globbing

while ( $nextname = </etc/host*>)  # Globbing im Skalaren Kontext
{
     print  “ Eine der Dateien ist $nextname \n“;
}
```

- Ein Wechsel von Verzeichnissen wird durch die Funktion chdir erreicht
- **Globbing:** Werden Argumente wie * oder /etc/host* zu einer Liste von passenden Dateinamen erweitert, so nennt man das Globbing
- In Perl erreicht man Globbing indem man das Muster in spitze Klammern setzt
- In einem skalaren Kontext liefert der Ausdruck so lange einen neuen Filenamen, bis keine Übereinstimmung mehr gefunden wird;

Umgang mit Directorys

```
opendir ( ETC, “/etc“) or die “/etc nicht vorhanden“;
while ( $name = readdir(ETC) ) { # skalarer Kontext, ein Name pro Durchlauf
      print “$name\n“; # Ausgabe von ...
}
closedir (ETC);

opendir ( ETC, “/etc“) or die “no /etc? “;
foreach   $name ( sort readdir(ETC) ) { # Array Kontext
      print “$name\n“; # Ausgabe von ...
}
closedir (ETC);
```

- Öffnen eines Verzeichnisses erfolgt mit opendir
- Lesen eines Directoyhandles erfolgt mit readdir
- Schließen eines Verzeichnisses erfolgt mit closedir
- Das Löschen eines Verzeichnisses erfolgt mit rmdir
- Mit mkdir kann man ein neues Verzeichnis anlegen

Löschen und Rename

```
print “welche Datei wollen Sie löschen?  “;
chomp ( $name = <STDIN> );
unlink ( $name );

unlink ( <*.o> );   # Alle Files mit Endung .o löschen

rename ( “fred“, “barney“) or die “fred kann nicht umbenannt werden“;

chmod 0755, @executables; # neuen mode setzen
```

- Das löschen von Files erfolgt mit unlink
- Ein Umbennenen kann mit rename erfolgen
- chmod ändert die Berechtigungen für ein File
- chown ändert den Benutzer eines Files

File Utilities

```
use File::Copy;
copy( “Quelldatei“, “Zieldatei“);

use File::Find;
find(\&wanted, ‘dir1‘, ‘dir2‘, ....);
# oder finddepth um unterste zuerst zu finden
sub wanted { ...}
# $File::Find::dir enthält aktuellen Verzeichnisnamen
# $_ enthält aktuellen Filenamen
# $File:Find:name enthält kompletten Namen mit Verzeichnis
```

- Die File Utilities sind Bibliotheksroutinen und müssen mit einem use Befehl geladen werden
- File::Copy kopiert Files
- File::Find durchsucht rekursiv Verzeichnisse nach Files

File::Path Utilities

```
use File::Path;
mkpath( [ '/hugo/otto/emil', '/franz/fritz/kuno'] , 1, 0711);
rmtree( [ '/hugo/otto/emil', '/franz/fritz/kuno'] , 1, 1);
```

- mkpath : Verzeichnisse erzeugen

  Argumente:

  - ⇨ Namen des zu erzeugenden Pfades oder Referenz auf eine Liste von Pfaden
  - ⇨ Boolscher Wert, der angibt ob mkpath den Namen des Verzeichnisses ausgeben soll, wenn es erzeugt ist
  - ⇨ Numerischer Modus bei der Erzeugung der Verzeichnisse (0777 default)

- rmtree : Bäume löschen

  Argumente:

  - ⇨ Das Stammverzeichnis oder eine Liste von Stammverzeichnissen. Alle Dateien darunter und das Stammverzeichnis werden gelöscht
  - ⇨ Boolscher Wert, der angibt ob rmtree eine Meldung ausgibt
  - ⇨ Ein Boolscher Wert, der angibt, ob rmtree alle Dateien überspringt bei denen kein Schreibrecht existiert

File::Basename

```
use File::Basename;

($name,$path,$suffix) = fileparse($fullname,@suffixlist);
$name = fileparse($fullname,@suffixlist);
fileparse_set_fstype($os_string);
$basename = basename($fullname,@suffixlist);
$dirname = dirname($fullname);

($name,$path,$suffix) = fileparse("lib/File/Basename.pm",qr{\.pm});
fileparse_set_fstype("VMS");
$basename = basename("lib/File/Basename.pm",".pm");
$dirname = dirname("lib/File/Basename.pm");
```

- Die File Basename Utilities erlauben die Zerlegung von Filenamen in verschiedenen Betriebssystemen
- Die Suffixliste ist eine Liste von einem oder mehreren regulären Mustern für die Endungen

File::Temp

```
use File::Temp qw/ tempfile tempdir /;

$fh = tempfile();
($fh, $filename) = tempfile();

($fh, $filename) = tempfile( $template, DIR => $dir);
($fh, $filename) = tempfile( $template, SUFFIX => '.dat');

$dir = tempdir( CLEANUP => 1 );
($fh, $filename) = tempfile( DIR => $dir );
```

- Temporäre Files erzeugen und löschen
- $template ist ein Name mit mindestens 4 X am Ende, z.B. "TempXXXX" oder "MyTempXXXX"

Hochschule Darmstadt
Fachbereich Informatik

Systemprogrammierung mit Perl

11. Der Perl Debugger

Der Perl Debugger

Kontrolle der Script Ausführung	
T	Stack trace
s [expr]	Single Step [in expr]
n [expr]	Next, springt über Unterprogramme
<CR/Enter>	Wiederhole letztes n oder s
r	Return from subroutine
c [ln|sub]	Continue until position
L	List break/watch/actions
t [expr]	Toggle Trace [trace expr]
b [ln|event|sub] [cnd]	Setze breakpoint
B ln|*	Lösche einen/alle Breakpoints
a [ln] cmd	Führe cmd vor Zeile aus
A ln|*	Lösche eine/alle Aktionen
w expr	Füge einen Watch Ausdruck hinzu
W expr|*	Lösche einen/alle Watch Ausdrücke
![!] syscmd	Führe syscmd in einem Subprocess aus
R	Versuche einen Neustart
q or ^D	Quit

Perl Debugger: Daten Untersuchung

Daten Untersuchung	
expr	Führe Perl Codeaus, siehe auch: s,n,t expr
x|m expr	Evaluiert expr in einem Listen Kontext, druckt das Ergebnis oder listet Methoden.
p expr	print expression (Benutzt script's augenblicklichliches package).
S [[!]pat]	Listet Subroutine Namen die [nicht] mit dem Muster übereinstimmen
V [Pk [Vars]]	Listet Variablen in dem Package. Vars kann sein: ~pattern oder !pattern.
X [Vars]	Das gleiche wie "V current_package [Vars]".
y [n [Vars]]	Listet Lexikalische Variablen in höherem scope <n>. Vars ist das gleiche wie bei V.

Perl Debugger: Quellcode

Liste/Suche Quellcode Zeilen	
l [ln|sub]	Liste Quellcode
- or .	Liste vorige/augenblickliche Zeile
v [line]	Zeige die Umgebung der Zeile
f filename	Zeige den Quellcode im File
/pattern/ ?patt?	Suche Vorwärts/Rückwärts
M	Zeige die Modulversion

Debugger Kontrolle

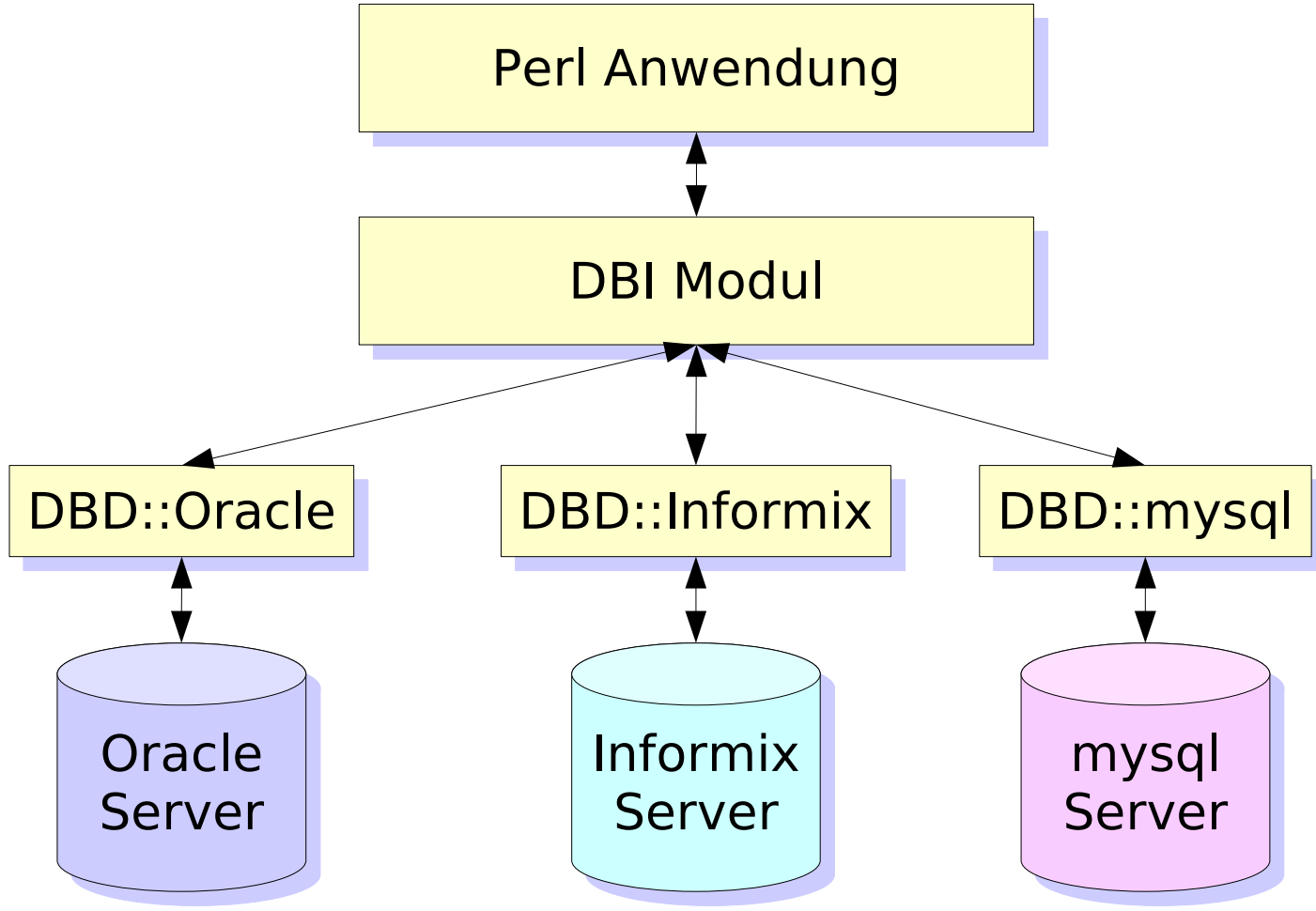
Debugger Kontrolle	
o [...]	Setze Debugger Optionen
<[<]|{[{]|>[>] [cmd]	Setze oder zeige pre/post-prompt Kommandos
! [N|pat]	Wiederhole ein vorangegangenes Kommando
H [-num]	Zeige die letzten num Kommandos
= [a val]	Definiere/liste einen alias
h [db_cmd]	Zeige Hilfe für ein Debug Kommando
h h	Zeige vollständige Hilfeseite
|[|]db_cmd	Sende den Output an einen Pager

Hochschule Darmstadt
Fachbereich Informatik

Systemprogrammierung mit Perl

12. Das Datenbank Interface DBI

Das DBI Modul als Vermittler



Aufbau einer Verbindung

```
#!/usr/bin/perl
# connect.pl - connect to the MySQL server

use strict;
use warnings;
use DBI;

my $dsn = "DBI:mysql:host=localhost;database=zettelkasten";
my $dbh = DBI->connect ($dsn, "ZettelUser", '#notiz?')
          or die "Cannot connect to server\n";

print "Connected\n";
$dbh->disconnect ();
print "Disconnected\n";
```

- für $dsn kann auch genutzt wrden, dass der default für host localhost ist
  - ⇨ $dsn = "DBI:mysql:zettelkasten";

Nutzung einer Bibliotheksfunktion und Abfangen des Fehlers durch eval

```
#!/usr/bin/perl

use strict;
use warnings;
use Zettelkasten;

my $dbh;
eval
{
      $dbh = Zettelkasten::connect ();
      print "Connected\n";
};
die "$@" if $@;
$dbh->disconnect ();
print "Disconnected\n";
```

- Insbesondere bei CGI Modulen sollte der Verbindungsaufbau in ein BibliotheksModul ausgelagert werden, da sonst das Passwort nicht gesichert ist

Bibliotheksfunktion

```
package Zettelkasten;
use strict;
use warnings;
use DBI;

my $db_name = "zettelkasten";
my $host_name = "localhost";
my $user_name = "ZettelUser";
my $password = "#notiz?";
my $port = undef;

# Establish a connection to the zettelkasten database, returning a database
# handle.  Raise an exception if the connection cannot be established.
sub connect
{
      my $dsn = "DBI:mysql:host=$host_name";
      my %conn_attrs = (PrintError => 0, RaiseError => 1, AutoCommit => 1);
      $dsn .= ";database=$db_name" if defined $db_name;
      $dsn .= ";port=$port" if defined $port;
      return (DBI->connect ($dsn, $user_name, $password, \%conn_attrs));
}
1;  # return true
```

Datenzugriff mit Array

```
#!/usr/bin/perl
use strict;
use warnings;
use Zettelkasten;

my $dbh = Zettelkasten::connect ();

{ # begin scope
      print "Fetch rows with fetchrow_array()\n";

      my $sth = $dbh->prepare ("SELECT id, Titel, Text  FROM notiz");
      $sth->execute () or die $dbh->errstr;
      my $count = 0;
      while (my @val = $sth->fetchrow_array ())
      {
            print "id: $val[0], Titel: $val[1], Text: $val[2]\n";
            ++$count;
      }
      $sth->finish ();
      print "Number of rows returned: $count\n";
} # end scope
```

Datenzugriff mit Arrayreferenz

```
{ # begin scope
      print "Fetch rows with fetchrow_arrayref()\n";

      my $sth = $dbh->prepare ("SELECT id, Titel, Text  FROM notiz");
      $sth->execute ();
      my $count = 0;
      while (my $ref = $sth->fetchrow_arrayref ())
      {
            print "id: $ref->[0], Titel: $ref->[1], Text: $ref->[2]\n";
             ++$count;
      }
      print "Number of rows returned: $count\n";
} # end scope
```

Datenzugriff mit Hash Referenz

```
{ # begin scope
      print "Fetch rows with fetchrow_hashref()\n";

      my $sth = $dbh->prepare ("SELECT id, Titel, Text  FROM notiz");
      $sth->execute ();
      my $count = 0;
      while (my $ref = $sth->fetchrow_hashref ())
      {
            print "id: $ref->{id}, Titel: $ref->{Titel}, Text: $ref->{Text}\n";
            ++$count;
      }
      print "Number of rows returned: $count\n";
} # end scope
```

Direkter Zugriff mit selectrow

```
{ # begin scope
      print "Fetch rows with selectrow_array()\n";

      my @val = $dbh->selectrow_array ("SELECT id, Titel, Text  FROM notiz
                                  WHERE id = 2");

      my $_count = $dbh->selectrow_array ("SELECT COUNT(*) FROM notiz");

      print "row: @val\n";
      print "count: $_count\n";
} # end scope
```

Update

```
{ # begin scope
     print "Update rows\n";

     my $count = $dbh->do ("UPDATE notiz SET Titel = 'Neuer Text'
                      WHERE Titel = 'Mein Titel3'");
     if ($count)   # print row count if no error occurred
     {
          $count += 0;
          print "Number of rows updated: $count\n";
     }
} # end scope
```

- Der Rückgabewert des DBI::do Aufrufs ist die Anzahl geänderter Zeilen
- Bei 0 geändertern Zeilen gibt das DBI Modul den Wert 0E0 zurück. Dieser Wert ist im numerischen Kontext 0 aber gleichzeitig wahr, da die Stringpräsentation nicht "0" ist.
- Die do Methode wird angewandt für einmalige updates und inserts

Update, vereinfacht

```
{ # begin scope

    my $count = $dbh->do ("UPDATE notiz SET Titel = 'Neuer Text'
                           WHERE Titel = 'Mein Titel3'");
    printf "Number of rows updated: %d\n", $count;

} # end scope
```

- Die do Methode stellt eine Kombination aus prepare und execute zur Verfügung und ist daher für einen UPDATE Zugriff gut geignet.
- printf stellt für count aufgrund des %d Formats einen numerischen Kontext zur Verfügung

INSERT mit Placeholdern

```
{ # begin scope
    print "Execute INSERT statement with do()\n";
    my $count = $dbh->do ("INSERT INTO notiz (Titel,Text,CDatum) VALUES(?,?,?)",
                          undef,
                          "Mein Titel1", "Mein Text", undef);
    printf "Number of rows inserted: %d\n", $count;
} # end scope
```

- Die Syntax für die do Methode ist
  - ⇨ $rows = $dbh->do($statement, \%attr, @bind_values) or die ...
- Der 2. Parameter ist daher eine Referenz auf einen Hash mit Attributen oder undef

Insert mit prepare

```
{ # begin scope
     print "Execute INSERT statement with prepare() + execute()\n";
     my $sth = $dbh->prepare ("INSERT INTO notiz (Titel,Text,CDatum)
                       VALUES(?,?,?)");
     my $count = $sth->execute ( "Mein Titel2", "Mein Text",undef);
     printf "Number of rows inserted: %d\n", $count;
} # end scope
```

- Die Nutzung der prepare Methode ist wichtig, wenn die Operation öfters ausgeführt werden soll
- Die Placeholder werden automatisch vom dbi Modul gequotet (mit Quotes versehen und Sonderzeichen mit Escape geschützt)

Beispiele: Aufbau des SQL Strings mit sprintf, Einfügen eines Hashs

```
sub insert_hash {
      my ($table, $field_values) = @_;

      # sort to keep field order, and thus sql, stable for prepare_cached
      my @fields = sort keys %$field_values;
      my @values = @{$field_values}{@fields};

      my $sql = sprintf "insert into %s (%s) values (%s)",
                              $table, join(",", @fields), join(",", ("?")x@fields);
      my $sth = $dbh->prepare_cached($sql);
      return $sth->execute(@values);
}
```

Beispiel: Suchen nach mehereren Werten

```
sub search_hash {
    my ($table, $field_values) = @_;
    # sort to keep field order, and thus sql, stable for prepare_cached
    my @fields = sort keys %$field_values;
    my @values = @{$field_values}{@fields};
    my $qualifier = "";

    $qualifier = "where ".join(" and ", map { "$_=?" } @fields) if @fields;
    $sth = $dbh->prepare_cached("SELECT * FROM $table $qualifier");
    return $dbh->selectall_arrayref($sth, {}, @values);
}
```