
Perl Grundlagen

Dirk Rüdiger

dirk.ruediger@partmaster.de



Seminarfahrplan

1	Variablen, Datentypen und Operatoren	8
2	Übung	31
3	Kontrollstrukturen	32
4	Erweiterte Datentypen	47
5	Übung	56
6	Funktionen und Subroutinen	57
7	Übung	71
8	Arbeiten mit Dateien	72
9	Reguläre Ausdrücke	90
10	Übung	108
11	Packages und Module	109
12	Perl Tipps	117

Auf nach Perl

- Perl wurde 1986 von Larry Wall entwickelt, es fand mittlerweile viele Freunde und eine große Entwickler-Gemeinde
- Derzeit aktuelle Version: 5.10
- Distributionen sind verfügbar für eine Vielzahl von Betriebssystemen
- Das Hauptquartier: <http://www.Perl.com>
- Für Windows:
 - Cygwin Project
<http://www.cygwin.com/>
 - ActivePerl von ActiveState
<http://www.ActiveState.com/>

Erstellen von Programmen

- Perl ist eine Interpreter-Sprache: das Skript wird bei jedem Aufruf kompiliert und dann ausgeführt
- Funktionserweiterungen (auch neue Klassen) für viele Anwendungsgebiete können als Packages bereit gestellt und in andere Programme integriert werden.
Im Comprehensive Perl Archive Network steht bereits eine große Auswahl an Modulen bereit.
- Perl-Programme können mit jedem beliebigen Editor erstellt werden

Ausführen von Programmen

- So geht's immer:

```
perl skriptname.pl
```

- Unix

- Datei-Attribut “ausführbar” des Skripts setzen

```
chmod a+x skriptname.pl
```

und

- die Aufruf-Parameter in der ersten Zeile des Skripts setzen

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

Ausführen von Programmen

- Windows

- Zum Ausführen von Perl-Skripten muss die Dateierweiterung `.pl` zur Systemvariablen `PATHEXT` hinzugefügt werden

```
echo %PATHEXT%  
.pl ; .com ; .exe ; .bat ; .cmd
```

- `perl.exe` muss dem Dateityp `.pl` für die Aktion `open` zugeordnet sein
- Das wird bei ActivePerl automatisch erledigt

Ein einfaches Script: Unser “Hallo Welt”

- Die Anweisung

```
print "Hallo_Kurs!\n";
```

- wird in die Datei `hallo.pl` geschrieben und aufgerufen

```
perl hallo.pl
```

und ausgeführt

- oder direkt ausgeführt

```
perl -e 'print "Hallo_Kurs!\n";'
```

- oder komplizierter

```
perl -e '$hallo="Hallo_Kurs!"; print "$hallo\n";'
```

Variablen, Datentypen und Operatoren

- Eingebaute Datentypen
- Zahlen und Zeichenketten
- Operatoren
- Was ist wahr?

Variablen, Datentypen und Operatoren

- Variablen
 - Dienen zum Speichern von Werten
 - Haben einen eindeutigen Bezeichner
 - Sind Instanzen eines Datentyps
- Datentypen
 - Beschreiben die Art einer Variable
 - Geben an, welche Werte eine Variable annehmen darf
- Operatoren
 - Beschreiben Anweisungen zum Verarbeiten der Variablen

Eingebaute Datentypen

Scalar ist ein einzelner –skalarer– Wert (Zahl oder Zeichenkette)

\$wert

Variablen dieses Typs beginnen mit \$.

Array eine Liste von Werten, numerisch geordnet

@feld

Variablen dieses Typs beginnen mit @.

Hash eine Gruppe von Werten, durch Zeichenketten geordnet

%woerterbuch

Sie werden auch als assoziative Listen bezeichnet. Variablen dieses Typs beginnen mit %.

Skalare Datentypen

- Ein Skalar kann Inhalt beliebigen Typs enthalten

```
$antwort = 123;           # Integer
$pi = 3.14159265;       # Gleitkommazahl
$name = "Guenther";     # Zeichenkette
$b = "Er_ist_$name";   # Variablen werden aufgeloeset
$c = 'Er ist $name';    # Variablennamen bleiben erhalten
$PI = $pi;              # Inhalt einer anderen Variable
$A = $PI / 4 * $D**2;   # Ergebnis eines Ausdrucks
$sich = 'who am i';     # Programm-Ergebnis
$object = new Table();  # ein Objekt
```

Skalare Datentypen

- Uninitialisierte Variablen werden existent wie benötigt
 - Sie werden mit 0 oder "" initialisiert
 - Sie werden automatisch als String, Zahl oder Boolean definiert (0 ist `false`, alle anderen Werte `true`)
- Der Inhalt der Variable wird automatisch konvertiert

```
$camels = '123';           # String  
print $camels + 1, "\n"; # ergibt 124
```

Skalare Datentypen – Zahlen

- Unterscheidung von Ganzzahlen und Gleitkommazahlen
- Dezimal-, Hexadezimal- oder Oktaldarstellung sowie Exponentialschreibweise möglich

Beispiele:	5 5000	Ganzzahlen
	45.3 .5544	Gleitkommazahlen
	-45 -1.64	negative Zahlen
	2e5 -2.4e4 6e-3	Zahlen in Exponentialschreibweise
	0xFFFFFFFF 0x00F1	Hexadezimalzahlen
	0333 0421	Oktalzahlen

Skalare Datentypen – Zeichenketten

- Aneinanderreihung beliebiger Zeichen des Unicode-Zeichensatzes
- Können Ziffern, Buchstaben, Symbole enthalten
- Werden in doppelten (") oder einfachen (') Anführungszeichen eingeschlossen
- Text in doppelten Anführungszeichen wird interpoliert.
Zeichen mit besonderer Bedeutung (\, @, \$, %, &) müssen innerhalb doppelter Anführungszeichen mit vorangestelltem \ maskiert werden.
- Text in einfachen Anführungszeichen ist literal (so wie er dort steht).
'\n' bedeutet eine Zeichenkette mit Inhalt '\ ' und '\n'

Skalare Datentypen – Zeichenketten

- HERE-Documents

- erleichtern umfangreiche Text-Ausgabe
- funktioniert für jede Art der Text-Ausgabe
- enthaltene Variablen werden vor Ausgabe expandiert

```
$int = 5; $string = " Birnen";
```

```
%ich = (name => " Guenther" );
```

```
print <<ENDE;
```

Und der **Text** von `$ich{name}` wird nun ausgegeben.

Ebenso der Inhalt von `$int` und `$string`.

ENDE

- Entstehende Zeichenkette kann auch einer Variable zugewiesen werden

Vordefinierte und spezielle Variablen

@ARGV enthält die Kommandozeilen-Argumente für das Skript

\$ARGV enthält den Namen der aktuellen Datei

@_ Parameterfeld für Subroutinen

%ENV Enthält die aktuellen Umgebungsvariablen

\$? Status des zuletzt aufgerufenen '...'-Kommandos, Pipe-close oder system-Aufruf

\$/ der Eingabeseparator, voreingestellt auf Zeilenvorschub. Kann auch mehrere Zeichen enthalten

\$0 Name der Datei, welche das aktuelle Skript enthält

\$\$ Prozess-ID des aktuell ausgeführten Perl-Programms

Exkurs: Ausgeben von Werten

```
$int = 5; $string = "Tag";  
print 'Hallo Welt!';           # ohne Zeilenvorschub  
print "Hallo_Welt!";          # ohne Zeilenvorschub  
print "Hallo_Welt!\n";        # mit Zeilenvorschub  
print $int;                    # 5  
print $int ** 2;               # 25  
print $int , $string , "e";   # 5Tage  
print "$int_{$string}";       # 5 Tag  
print "$int_{$string}e";      # 5 Tage  
print '$int {$string}e';      # $int {$string}e
```

Operatoren

- Arithmetische Operatoren
- Vergleichsoperatoren
- Logische Operatoren
- Zuweisungsoperatoren
- Bitbearbeitungsoperatoren
- Zeichenkettenoperatoren

Arithmetische Operatoren

Operatoren für mathematische Berechnungen, übergebene Werte werden bei Bedarf automatisch konvertiert.

- Addition, Subtraktion, Multiplikation, Division

$\$c = \$a + \$b;$

$\$b -= 1;$

$\$c = \$a / \$b;$

$\$c *= 2;$

- Modulus (Rest der Division)

$\$c = \$a \% \$b;$

Arithmetische Operatoren

- Exponentierung (Potenz)

`$c = $a ** $b;`

- Autoinkrement und Autodekrement

`++$a; $a++;`

`--$a; $a--;`

Position des Operators (`++`, `--`) hat dabei Einfluss auf Ergebnis:

- Operator vor Variable erhöht Wert der Variable und gibt diesen Wert zurück.
- Operator nach Variable gibt den Wert zurück und erhöht dann Wert der Variable.

Vergleichsoperatoren

Vergleichsoperatoren dienen der Entscheidungsfindung und liefern ein logisches Ergebnis. Sie werden in Kontrollstrukturen verwendet.

- Es werden Vergleichsoperatoren für numerische Werte und Zeichenketten unterschieden.

- Gleichheit, Ungleichheit

`==, !=` # numerisch

`eq, ne` # String

- Kleiner/Größer als

`<, >` # numerisch

`lt, gt` # String

Vergleichsoperatoren

- Kleiner/Größer gleich

`<=, >=` # numerisch

`le, ge` # String

- Vergleich

`<=>` # numerisch

`cmp` # String

- Vergleich von Zeichenketten mit numerischen Operatoren vergleicht nicht den Text, sondern den automatisch konvertierten Zahlenwert.

```
$false = "123" eq "123.00" ;
```

```
$true  = "123" == "123.00" ;
```

Logische Operatoren

Logische Operatoren testen die “Wahrheit” von Werten.

- **Nicht:** Umkehrung des Wahrheitswertes

`$b = !$a ;`

- **Und:** Ausdruck ist wahr, wenn beide Operanden wahr sind

`$c = $a && $b ;`

`$c = $a and $b ;`

- **Oder:** Ausdruck ist wahr, wenn einer der Operanden wahr sind

`$c = $a || $b ;`

`$c = $a or $b ;`

Bitbearbeitungsoperatoren

Bei diesen Operatoren werden die Operanden als Folge von Binärwerten (0 oder 1) verarbeitet.

- Bitweise Nicht: alle Bits werden invertiert

$$c = \sim b;$$

- Bitweise Oder: alle Bits werden nach Oder-Logik verknüpft

$$c = a | b;$$

- Bitweise Und: alle Bits werden nach Und-Logik verknüpft

$$c = a \& b;$$

Bitbearbeitungsoperatoren

- Bitweise exklusives Oder: alle Bits werden nach Xoder-Logik verknüpft

$$c = a \oplus b;$$

- Bitweise Links- oder Rechts-Verschiebung: alle Bits werden nach links oder rechts verschoben, Auffüllen mit 0

$$c = a \ll 1; \# \text{ Linksverschiebung}$$
$$d = a \gg 1; \# \text{ Rechts-Verschiebung}$$

Zeichenkettenoperatoren

Zeichenkettenoperatoren stellen spezielle Möglichkeiten zum Verarbeiten von Zeichenketten bereit.

```
$a = "Sonn" ; $b = "tag" ;
```

- Zusammenfügen (Stringaddition)

```
$so = $a . $b ;      #"Sonntag"
```

- Duplizieren (Stringmultiplikation)

```
$c = $b x 2 ;      #"tagtag"
```

Quote und Quote-like Operatoren

- Quote-Operatoren bieten verschiedene Möglichkeiten zu Interpolation und Pattern-Matching an

Angepasst	Ursprünglich	Bedeutung	Interpolation
' '	q{}	Literal	Nein
" "	qq{}	Literal	Ja
' '	qx{}	Systemkommando	Ja (außer ' ' ist Quotierungszeichen)
	qw{}	Wortliste	Nein
//	m{}	Pattern match	Ja (außer ' ' ist Quotierungszeichen)
	s{}{}	Substitution	Ja (außer ' ' ist Quotierungszeichen)
	tr{}{}	Transliteration	Nein

- Es können (nahezu) beliebige Quotierungszeichen genutzt werden
- Nicht-Klammern als Quotierungszeichen nehmen das gleiche Zeichen vor und nach dem Ausdruck. Beide Ausdrücke sind identisch:

```
q{ foo { bar } baz }  
' foo { bar } baz '
```

Zuweisungsoperatoren

Zuweisungsoperatoren gestatten das Übergeben des rechtseitigen Ausdrucks an die Variable auf der linken Seite.

- `lvalue = rvalue`

```
$a = $b;
```

```
$a = $b + 5;
```

- `lvalue = lvalue op rvalue`

```
$a = $a + 3;
```

- `lvalue op= rvalue`

```
$a += 3;
```

```
$so .= " _list_frei";
```

Priorität der Operatoren

++, --

**

! ~

* / % x

+ - .

<< >>

< <= >= > lt gt le ge

= != <=> eq ne cmp

&

| ^

&&

||

Ausdrücke können beliebig durch Klammern gruppiert werden.

Exkurs: Was ist wahr?

Wahr ist

- Jeder String, außer "" und "0"
- Jede Zahl, außer 0
- Jede Referenz

1

"0.00"

\\$value

"Hallo"

12345667

0.00000000000000000001

Falsch ist

- Der String "" und "0"
- Die Zahl 0
- Jeder undefinierte Wert

10 - 10

0.00

"0"

""

"0.00" + 0

undef()

Hinweis: "0.00" ist eine Zeichenkette mit 4 Zeichen und ist damit *wahr!*

Übung

Kontrollstrukturen

- Anweisungen zur Bedingungsauswahl
- Wiederholungsschleifen

Kontrollstrukturen

- Dienen zur Einflussnahme auf den Programmablauf
- Ausführung von Anweisungsblöcken wird durch eine definierte Entscheidung bestimmt, deren Auswertung das Ergebnis `wahr` liefern muss.
- Bedingungen werden in Klammern (und) zusammengefasst und gruppiert
- Anweisungsblöcke werden von geschweiften Klammern { und } eingeschlossen

Anweisungen zur Bedingungs Auswahl

Unterscheidung der Auswahlkonstrukte erfolgt nach der Anzahl der wählbaren Alternativen

Auswahl	Anzahl der Möglichkeiten	Anweisung
Einseitig	eine	if, unless
Zweiseitig	zwei	if-else, unless-else, .. ? .. : ..
Mehrstufig	mehrere (>2)	if-elsif-else

Anweisung zur Bedingungsauswahl – Einseitig

- Ein Anweisungsblock wird ausgeführt, wenn die Bedingung erfüllt ist.

- If-Anweisung

```
if ($a > 1) {  
    print "\$a > 1\n";  
}
```

- Unless-Anweisung

```
unless ($a > 1) {  
    print "\$a <= 1\n";  
}
```

Anweisung zur Bedingungsauswahl – Zweiseitig

- Wenn die Bedingung erfüllt ist, wird der eine Anweisungsblock ausgeführt, andernfalls wird der alternative Anweisungsblock ausgeführt.
- If-Else-Anweisung

```
if ($a > 1) {  
    print "\$a > 1\n";  
}  
else {  
    print "\$a <= 1\n";  
}
```

Anweisung zur Bedingungsauswahl – Zweiseitig

- Unless-Else-Anweisung

```
unless ($a > 1) {  
    print "\$a <= 1\n";  
}  
else {  
    print "\$a > 1\n";  
}
```

- Eine Kurzform der zweiseitigen Auswahl bietet der ternäre Vergleichsoperator

(Bedingung) ? Operand1 : Operand2

Anweisung zur Bedingungsauswahl – Mehrstufig

- Es existieren mehrere Anweisungsblöcke, von denen ein Block je nach der definierten Bedingung ausgewählt wird.
- If-Elself-Else-Anweisung

```
if ($a > 1) {  
    print "\$a > 1\n";  
}  
elseif ($a == 1) {  
    print "\$a == 1\n";  
}  
elseif ($a <= 10) {  
    print "\$a <= 10\n";  
}  
else { print "\$a < 10\n"; }
```

Wiederholungsschleifen

- Unterscheidung der Wiederholungsschleifen nach
 - zahlenmäßig bekannter oder
 - von Bedingung abhängigerAnzahl der Wiederholungsschleifen.
- Bedingte Wiederholungen können je nach der Anordnung der Bedingung
 - kopfgesteuert (Auswertung der Bedingung vor Schleifendurchlauf) bzw.
 - fußgesteuert (Auswertung der Bedingung nach Schleifendurchlauf)sein.

Wiederholungsschleifen – bedingt, kopfgesteuert

- While-Anweisung

```
while ($i < 10) {  
    print $i++ . "\n";  
}  
print "Fertig!" ;
```

```
while (@tage) {  
    print (shift @tage) . "\n";  
}
```

Wiederholungsschleifen – bedingt, kopfgesteuert

- Until-Anweisung

```
until ($i == 10) {  
    print $i++ . "\n";  
}  
print "Fertig!" ;
```

Wiederholungsschleifen – bedingt, fußgesteuert

- While-Anweisung

```
do {  
    print $i++ . "\n";  
} while ($i < 10)
```

- Until-Anweisung

```
do {  
    print $i++ . "\n";  
} until ($i == 10)
```

Wiederholungsschleifen – zählergesteuert

- For-Anweisung

```
for ($i=0; $i<10 ; $i++) {  
    print "$i\n";  
}
```

```
for (;;) {  
    print "Hallo _Du!";  
}
```

Wiederholungsschleifen – zählergesteuert

- Foreach-Anweisung

```
foreach my $tag (@tage) {  
    print $tag;  
}
```

```
my $i = 0;  
foreach my $element (sort keys %hash) {  
    print "$i: _$hash{$element}"; $i++;  
}
```

Wiederholungsschleifen – Eingriff in den Schleifenlauf

Perl kennt verschiedene Anweisungen zur Beeinflussung des Schleifendurchlaufs.

- `redo` Abbruch des aktuellen Schleifendurchlaufs und Neubeginn, ohne die Schleifenbedingung erneut auszuwerten
- `next` Abbruch des aktuellen Schleifendurchlaufs und Neubeginn einschließlich dem Auswerten der Schleifenbedingung
- `last` Sofortiger Abbruch des Schleifendurchlaufs
- `exit` Beendet die Ausführung des Programmes

Wiederholungsschleifen – Eingriff in den Schleifenlauf

```
foreach my $user (@users) {  
    next if ($user eq "root");  
    if ($user eq "drue") {  
        print "Found drue!\n";  
        last;  
    }  
}
```

Erweiterte Datentypen

- Listen und Arrays
- Hashes
- Slices

Erweiterte Datentypen

- Unterscheidung zwischen Arrays und Hashes.
- Beide dienen zur Speicherung einer größeren Anzahl logisch zusammengehöriger Werte.
- Mittels Wiederholungsschleifen kann über die Elemente von Arrays und Hashes iteriert werden.
- Es kann auf die einzelnen Elemente direkt zugegriffen werden.

Listen und Arrays

- Eine Liste ist eine Sammlung von Skalaren.
- Ein Array ist der Perl-Datentyp, in dem Listen gespeichert werden können.
- Auf Werte eines Arrays kann über deren Position zugegriffen werden, Startwert ist 0

```
@num = (1, 2, 3, 4, 5);  
print "$num[0] _$num[1] _$num[2] _$num[3]\n"; # 1 2 3 4
```

- Arrays können Zahlen, Zeichenketten oder einen Mix aus Beidem enthalten

```
@fuzz = ("Guenther", 0, "Kasten", 4, "Tiger");
```
- Array-Variablen haben einen eigenen Namensraum, die Namen beginnen immer mit @
- Es können auch ganze Listen als Wert eines Listen-Elements angelegt werden: `$array[index]=\@anderes_array`

Listen und Arrays

```
my @tage = ("Mo" , " Di" , " Mi" , " Do" , " fr" );
$tage[5] = " Sa" ;           # Wert anhaengen
$tage[4] = " Fr" ;         # Wert aendern
my $sa = pop(@tage);       # letzten Wert holen
push(@tage , $sa );        # Werte anhaengen
my $mo = shift(@tage);     # ersten Wert holen
unshift(@tage , $mo);      # ersten Wert einfuegen
my @woche = (@tage , " So" ); # Neue Liste erzeugen
print $tage[-1];           # zeige letztes Element
print $tage[3,4,5];        # zeige Element 4-6
print $#tage                # hoechsten Index des Arrays
```

Intervall-Operator in Listen

- dienen zur Vereinfachung der Definition von Intervallen in Listen
- Intervall-Operator ist ..
- Intervall-Operator kann in numerischen und alphabetischen Listen benutzt werden
- Beispiele:

(1, 2, 3, 4, 5, 6, 7)	=	(1 .. 7)
(2, 4, 6, 7, 8, 9, 10, 12)	=	(2,4,6 .. 10, 12)
(a,b,c,d,e,f)	=	(a .. f)
(1.5, 2.5, 3.5, 4.5)	=	(1.5 .. 4.5)
(-3, -2, -1, 0, 1, 2, 3)	=	(-3 .. 3)
(1, 2, 3, 4, 20, 21, 22)	=	(1 .. 4, 20 .. 22)

Hash

- Ungeordnete Liste von Skalaren (assoziatives Array)
- Hash-Variablen haben einen eigenen Namensraum, die Namen beginnen immer mit %
- Auf die Elemente eines Hashes kann über deren alphanumerischen Schlüsselwert zugegriffen werden: `$hash{key}`
- Alle Einträge bestehen aus einem Tupel von `key` und `value`

```
my %person = (name => "Guenther", alter => 1);
```
- Es können auch ganze Listen oder Hashes als Wert eines Hash-Elements angelegt werden: `$hash{index}=\%anderes_hash`

Hash

```
my %freitage = ("Sa"=>"samstag" ,"So"=>"Sonntag" );
$freitage {'Sa'} = "samstag" ;
print " Bald_ist_.$freitage {'So'}.\n" ; # Bald ist Sonntag.
```

```
my @schluessel = keys %glossar ; # Array-Kontext
my @werte      = values %glossar ;
my $anzahl     = keys %glossar ; # skalarer Kontext!
print " Das_Glossar_hat_.$anzahl_Eintraege.\n" ;
```

```
delete $glossar {"Lehrer"} ; # Loeschen eines Eintrags
exists $glossar {"Lehrer"} ; # wahr, wenn Element existiert
my ($key, $val) = each %glossar ; # Schluessel-Wert-Paar
```

Anwendung von Wiederholungsschleifen – foreach-Anweisung

- Array

```
foreach my $value (@array) {  
    print "$value\n";  
}  
foreach (@array) { print $_; } # Kurzform!
```

- Hash

```
foreach my $value (sort keys %hash) {  
    print "$hash{$value}\n";  
}
```

Slices

- Bequemer Weg zum Zugriff auf mehrere Elemente eines Arrays
- Elemente werden als Liste im Subskript angegeben

```
my ($him, $her) = @folks[0, -1];  
my @them       = @folks[0 .. 3];  
my ($uid, $dir) = (getpwnam("daemon"))[2, 7];
```

- Ergebnis kann auch einem Slice zugewiesen werden

```
my @days[3..5] = qw/Wed Thu Fri /;  
my @folks[0, -1] = @folks[-1, 0];
```

- Ändern eines Slices ändert die originalen Array- oder Hash-Elemente

```
foreach my $value (@array[ 4 .. 10 ]) {  
    $value =~ s/peter/paul /;  
}
```

Übung

Funktionen und Subroutinen

- Eingebaute und selbst-definierte Funktionen
- Perl's eingebaute Funktionen
- Eigenschaften von Funktionen
- Definition von neuen Funktionen

Subroutinen in Perl

- Subroutinen sind “benannte” Anweisungsblöcke
- Sie können vom Hauptprogramm oder anderen Subroutinen aufgerufen werden
- Sie dienen zur Erhaltung der Übersichtlichkeit der Programmstruktur
- Perl unterscheidet syntaktisch nicht zwischen den Bezeichnern Unterprogrammen, Funktionen, Subroutinen, Prozeduren etc. – alle Bezeichnungen können synonym benutzt werden
- Subroutinen haben einen eigenen Namensraum, die Namen beginnen immer mit & (kann bei Eindeutigkeit weg gelassen werden)

Subroutinen in Perl

- An die Subroutine können beliebig viele Parameter in Form einer Argumentliste übergeben werden. Somit kann die Parameterübergabe sehr flexibel gestaltet werden

```
print $name, $alter, @kinder, "\n";  
print $name, $alter, "\n";
```

- In der Subroutine enthält
 - `@_` eine Liste aller Argumente der Funktion
 - `$_` jeweils das erste Element der Argumentliste
- Subroutinen können ein Ergebnis an das aufrufende Programm zurück geben, dies erfolgt mit der Anweisung

```
return $wert; # Wert kann von beliebigem Typ sein!
```

Subroutinen in Perl

- Der Aufruf einer Subroutine kann bereits vor dessen Deklaration erfolgen
- Eingebaute Funktionen (Befehle) gehören zum Sprachumfang von Perl, z.B.

`print , pop , exp , open , shift , return , split , chomp , etc .`

- nutzerdefinierte Funktionen (Subroutinen) können selbst definiert werden

```
sub suchdiewerte {  
    my $pattern = shift ;  
    return grep /$pattern/ @_ ;  
}
```

Subroutinen in Perl

```
sub splitValues ($) {
    my $arg = $_;           # alternativ: my $arg = shift;
    my ($key, $val) = split("=", $arg);
    $key =~ tr/A-Z/a-z/;
    return ($key, $val);
}

$string = "SchlueSSeL=Wert";
@feld = splitValues($string); # @feld = ("schluessel", "Wert")
```

Lokalisieren von Variablen

- Alle Variablen sind im globalen Kontext deklariert, sie sind im gesamten Programm sichtbar
- Deklarations-Präfixen `my` und `local` begrenzt Sichtbarkeit der Variable auf den Anweisungsblock

my – "lexically scoped Variable"

Variable ist nur in dem Anweisungsblock gültig

local – "dynamically scoped Variable":

Variable ist in dem Anweisungsblock und in allen von hier aufgerufenen Subroutinen gültig

- `my` ist `local` vorzuziehen - es ist schneller und sicherer.
- Deklarieren einer Liste von lokale Werte ist möglich

```
my ( $integer , $string , %hash , @array );
```

Lokalisieren von Variablen

- Beispiel: my

```
sub eineSubroutine {  
    my $arg = shift;  
    my $var = $arg / 6;  
    print "$var\n"; # 4  
}
```

```
my $var = 24;  
print "$var\n"; # 24  
eineSubroutine($var);  
print "$var\n"; # 24
```

Vordefinierte Funktionen

- Mathematische Funktionen

<code>abs(\$zahl)</code>	Liefert den Absolutwert
<code>int(\$zahl)</code>	Liefert den Ganzzahlanteil
<code>sin(\$zahl), cos(\$zahl)</code>	Winkelfunktionen
<code>exp(\$zahl)</code>	Exponentialfunktion
<code>log(\$zahl)</code>	Natürlicher Logarithmus einer Zahl
<code>sqrt(\$zahl)</code>	Gibt die Quadratwurzel zurück
<code>rand(\$zahl)</code>	Liefert eine Zufallszahl kleiner als \$zahl
<code>srand(\$zahl)</code>	Initialisiert den Zufallszahlengenerator
<code>time()</code>	Gibt die Sekunden seit dem 01.01.1970 zurück

Vordefinierte Funktionen

- Funktionen zur Zeichenkettenverarbeitung

`chomp($str)`

Löscht Zeilenumbruch am Ende der Zeichenkette

`chop($str)`

Löscht das letzte Zeichen einer Zeichenkette

`index($str, $s)`

Liefert die Position der Zeichenkette `$s` in `$str`

`rindex($str, $s)`

Liefert die letzte Position der Zeichenkette

`length($str)`

Liefert die Länge der Zeichenkette

`substr($str, $von, $len)`

Liefert eine Teil-Zeichenkette von `$str`

`split(/regex/, $str)`

Zerlegt eine Zeichenkette in ein Stringarray

`join($str, @array)`

Verbindet ein String-Array zu einer Zeichenkette

Vordefinierte Funktionen

- Funktionen zur Zeichenketten-Formatierung

`printf ("Format", @var)` Gibt Zeichenkette formatiert aus

`sprintf ("Format", @var)` Gibt Zeichenkette formatiert zurück

Angabe der Formatierung erfolgt typisiert: `%<format><typ>`

`%-20s` Zeichenkette, 20 Zeichen lang, linksbündig

`%03d` Ganzzahl, dreistellig mit 0 aufgefüllt

`%5.3f` Gleitkommazahl, 5-stellig, 3 Dezimalstellen

`%%` Prozentzeichen

```
printf " %5.3f%% des %s in KW %02d.\n" , 23.2 , " Bestands" , 7;  
# Ergibt: 23.200% des Bestands in KW 07.
```

Vordefinierte Funktionen

- Funktionen zur Zeichenketten-Manipulation

`$sstring =~ s/re/repl/;` Durchsucht `$sstring` nach `re` und ersetzt es durch `repl`.

`$sstring =~ tr/li1/li2/;` Tauscht in `$sstring` alle Elemente aus `li1` gegen `li2`.

- Funktionen zur Konvertierung

`localtime ($zeit)` Konvertiert Zeit in Zeichenkette oder Array

`chr($zahl)` Gibt das zu `$zahl` gehörende ASCII-Zeichen zurück

`hex($hexstring)` Gibt Dezimalwert des Hex-Strings `$hex` zurück

`oct($octstring)` Gibt Dezimalwert des Octal-Strings `$oct` zurück

Vordefinierte Funktionen

- Funktionen zur Verarbeitung von Hashes, Arrays und Listen

<code>delete \$hash{\$element}</code>	Löscht ein Datenelement
<code>each %hash</code>	Liste von zwei Elementen (key, val)
<code>grep EXPR, @list</code>	Liste aller auf EXPR passenden Elemente
<code>grep {block} @list</code>	Liste aller auf {block} passenden Elemente
<code>map EXPR, @list</code>	Wendet den Ausdruck EXPR auf alle Elemente an
<code>map {block} @list</code>	Wendet {block} auf alle Elemente an
<code>reverse @list</code>	Liste in umgekehrter Reihenfolge
<code>scalar @list</code>	Anzahl der Elemente zurück
<code>values %hash</code>	Liste aller Werte des Hashes

Vordefinierte Funktionen

- Dateioperationen

<code>stat \$fname</code>	Gibt Informationen über Datei zurück
<code>chmod modus, @dateien</code>	Ändert Zugriffsrechte für Dateien
<code>chown UID, GID, @dateien</code>	Ändert Besitzrechte für Dateien
<code>mkdir \$dname</code>	Legt das Verzeichnis an
<code>rmdir \$dname</code>	Löscht das Verzeichnis, wenn es leer ist
<code>rename \$datei1, \$datei2</code>	Benennt Datei um
<code>unlink @dateien</code>	Löscht Liste von Dateien
<code>link \$datei1, \$datei2</code>	Erzeugt Link von \$datei1 nach \$datei2
<code>symlink \$datei1, \$datei2</code>	Erzeugt einen symbolischen Link

Vordefinierte Funktionen

- Verzeichnis-Operationen

<code>opendir HANDLE, \$dir</code>	öffnet Verzeichnis und gibt Handle zurück
<code>readdir HANDLE</code>	nächsten Eintrag aus dem Verzeichnis
<code>rewinddir HANDLE</code>	Positioniert das Verzeichnis auf den Anfang
<code>telldir HANDLE</code>	aktuelle Position im Verzeichnis
<code>seekdir HANDLE, \$pos</code>	Setzt Position für <code>readdir</code> auf das Verzeichnis
<code>closedir HANDLE</code>	Schließt ein mit <code>opendir</code> geöffnetes Verzeichnis
<code>chdir \$dir</code>	Wechselt in das Verzeichnis

Übung

Arbeiten mit Dateien

- Zugriff auf Dateien
- Lesen und Schreiben von Datei-Inhalten
- Zugriff auf Verzeichnisse

Perl und Dateien

- Perl kennt viele Möglichkeiten, auf Dateisysteme und Dateien zuzugreifen
- Durch seine Herkunft als Werkzeug zur Report-Erzeugung ist dies neben der Stringverarbeitung ein Grundbaustein der Sprache
- Zugriff auf Dateien erfolgt bei allen Betriebssystemen in gleicher Weise
- Es wird nicht zwischen dem Zugriff auf Dateien oder andere Prozesse unterschieden
- Das Windows-Verzeichnis-Trennzeichen “\” muss wegen seiner Sonderbedeutung maskiert (“\\”) oder als “/” geschrieben werden
- Wenn vom Betriebssystem unterstützt, dann wird bei den Dateinamen zwischen Groß- und Kleinschreibung unterschieden

Öffnen von Dateien

- Zugriff auf Dateien erfolgt über Datei-Deskriptor (Filehandle)
- Erzeugen eines Datei-Deskriptors (Filehandle) mit der Funktion `open(my $fhandle, "$<$accesstype$>$", $dateiname)`
- Aufruf von `open` gibt `true` bei erfolgreicher Ausführung zurück
- Durch vor- oder nachgesetzte Zeichen wird der Zugriffsmodus spezifiziert

```
open(my $f, "text.txt");           # Aus Datei lesen ...
```

```
open(my $f, "<", "text.txt");      # ... ebenso
```

```
open(my $f, ">", "text.txt");      # Datei erstellen
```

```
open(my $f, ">>", "text.txt");     # an Datei anhaengen
```

```
open(my $f, "|program");           # an anderes Programm uebergeben
```

```
open(my $f, "program|");           # aus anderem Programm uebernehm
```

Öffnen von Dateien

- Explizites Öffnen von Dateien mit `sysopen()`
 - Zugriffsmodus wird konkret angegeben
 - Wird genutzt
 - * zur feinkörnigen Zugriffssteuerung
 - * bei Dateinamen, die `open()` nicht zulassen
 - Modus wird durch OR-Verknüpfung von Attributen definiert
 - Wenn Permission nicht angegeben, dann wird 0666 gesetzt

```
my $fname = "+a.txt";  
my $flags = O_RDWR|O_CREAT;  
my $perm = 0755;  
sysopen(FH, $fname, $flags);  
sysopen(FH, $fname, $flags, $perm);
```

Öffnen von Dateien

- Erfolgreiche Ausführung von `open()` und `sysopen()` kann geprüft werden
 - die `@msg` Gibt Fehlermeldung aus und beendet die Programmausführung
 - `warn @msg` Gibt Fehlermeldung aus und setzt Programmausführung fort

```
$file = "input.txt";  
open(my $dat, "<", $file) or warn "Couldn't open $file: $!\n";  
  
$file = "report.txt";  
open(my $dat, ">", $file) or die "Couldn't open $file: $!\n";
```

- Bei Modulen besser `carp()` und `croak()` aus dem Carp-Modul einsetzen

Öffnen von Dateien mit IO::File

- Modul IO::File stellt Objekt-Methoden für Datei-Deskriptoren bereit
- erlaubt Aufruf-Syntax wie open() und sysopen() sowie (ANSI-C) fopen(3)
- open()-Methode erzeugt Objekt

```
use IO::File;  
my $fh = new IO::File;  
if ($fh->open("<_file")) {  
    print <$fh>;  
    $fh->close;  
}
```

Zugriffsmodi und deren Notation

Filename	Read	Write	Append	Create	Trunc	O_Flags
<file	yes	no	no	no	no	RDONLY
>file	no	yes	no	yes	yes	WRONLY TRUNC CREAT
>>file	no	yes	yes	yes	no	WRONLY APPEND CREAT
+ <file	yes	yes	no	no	no	RDWR
> +file	yes	yes	no	yes	yes	RDWR TRUNC CREAT
+ >>file	yes	yes	yes	yes	no	RDWR APPEND CREAT

- > + und + >> sollte **nie** genommen werden: > + schneidet die Datei ab, bevor sie gelesen wird, + >> ist verwirrend, da der Lesezeiger irgendwo sein kann (wird aber oft vom System beim Schreiben ans Dateiende gesetzt).
- Die Verfügbarkeit der O_*-Konstanten hängt vom Betriebssystem ab – siehe open(2)
- O_*-Konstanten werden im Modul Fcntl.pm definiert.

Lesen aus Dateien

- Zeilenweise: Ein definierter Datei-Deskriptor wird in spitzen Klammern <> angegeben und wie ein Ausdruck gehandhabt

```
my $zeile = <DAT>; # eine Zeile lesen
```

```
my @zeilen = <DAT>; # alle Zeilen lesen
```

```
while (my $line = <STDIN>) { # Iterieren ueber Standard-Eingabe
    chomp $line;
    print "$line\n";
}
```

Lesen aus Dateien

- Zeichenweise: Zum Verarbeiten binärer Dateien
 - wenn Struktur nicht zeilenweise
 - Steuerzeichen und Zeilenwechsel haben keine Sonderbedeutung
 - Es kann ein einzelnes oder beliebig viele Zeichen gelesen werden
 - `binmode(FH)` wechselt von Text- und Binärmodus, wenn das Betriebssystem dies unterscheidet.
 - Eingestellter Binärmodus (`binmode`) bleibt bis zum Schließen der Datei erhalten.

Lesen aus Dateien

- Dateizeiger positionieren und prüfen

```
seek(FH, $position, $offset) or die "Error: _$!\n";  
$position = tell(FH); # position from file start
```

- Position ist abhängig vom Offset

- 0 Position von Dateianfang aus
- 1 Relative Verschiebung von aktueller Position
- 2 Position von Dateiende aus

Lesen aus Dateien

– Beispiel:

```
binmode(FH) if ($^O =~ /DOS|Win/);
#Datei: [abcd] Irgend_eine_80-Zeichen-Bytefolge
my $char = getc(FH);
if ($char eq "c" && !eof(FH)) {
    $chars = read(FH, $text, 80, 3);
    print "($chars)_", &strings($text), "\n";
}
```

Anmerkungen dazu:

- * `getc()` liest ein Zeichen und gibt es zurück
- * `read()` liest (beliebig viele) Zeichen in die Variable `$text` und gibt Anzahl der gelesenen Zeichen zurück

Schreiben in Dateien

Der Datei-Deskriptor wird in der `print`-Anweisung vor der Argumentliste angegeben

```
print HANDLE @msg;
```

Bei Ausgabe auf die Standard-Ausgabe kann der Datei-Deskriptor `STDOUT` entfallen.

```
print DAT "Hallo_Welt!"; # in Datei schreiben
```

```
while (my $zeile = <INFILE>) { # einfaches Kopierprogramm  
    print OFILE $zeile;  
}
```

Vordefinierte Datei-Deskriptoren

STDIN Standardeingabe

```
my $eingabe = <STDIN>;
```

STDOUT Standard-Ausgabe

```
print STDOUT "Hallo_Welt";
```

STDERR Standard-Fehlerausgabe

```
print STDERR "Hallo_Welt";
```

DATA Verweist auf alles, was im Programm nach der Zeile

```
__END__
```

folgt

Vordefinierte Datei-Deskriptoren

ARGV aktuell zu verarbeitende Datei der Kommandozeile

```
while (my $zeile = <ARGV>) {  
    my ($wert1, $wert2) = split (/:/, $zeile);  
}
```

und das lässt sich kurz schreiben:

```
while (my $zeile = <>) {  
    my ($wert1, $wert2) = split (/:/, $zeile);  
}
```

Wie funktioniert das nun?

```
open(my $f, ">", "datei.txt") or die "Fehler: _$!\n";
foreach my $tag (keys(%freitage)) {
    print $f "$tag=$freitage{$tag}\n"; # Schreiben
}
close $f;
open(my $g, "<", "datei.txt") or die "Fehler: _$!\n";
while (my $zeile = <$g>) { # Lesen
    chomp $zeile;
    my ($key, $val) = split("=", $zeile);
    $freitage{$key} = $val;
}
close $g;
```

Filetestoperatoren

- Dienen zum Ermitteln von Status-Informationen einer Datei
- Nutzbar in Bedingungen, schon bevor die Datei geöffnet wird
if (`-e $datei`) { ... }
- Ergebnis des Testoperators ist ein boolescher Wert

`-e $datei # Datei existiert`

`-r $datei # Datei ist lesbar`

`-w $datei # Datei ist schreibbar`

`-d $datei # Datei ist ein Verzeichnis`

`-f $datei # Datei ist eine Datei`

`-T $datei # Datei ist eine Textdatei`

`-z $datei # Datei hat Groesse 0`

Arbeiten mit Verzeichnissen

- In Unix werden
 - Dateinamen in Verzeichnissen organisiert
 - Informationen über Dateien (Größe, Besitzer, Zugriffsrechte, etc.) in Inodes gehalten
- Berechtigungen des Verzeichnisses bestimmen Zugriffsrechte auf Verzeichnis-Informationen
- Berechtigungen der Dateien bestimmen Zugriffsrechte auf deren Meta-Informationen
- `opendir()`, `readdir()`, `seekdir()`, `telldir()`, `rewinddir()` und `chdir()` gestatten Zugriff auf Verzeichnis-Informationen
- Verzeichnis-Einträge sind nicht alphabetisch geordnet.
- Achtung: Ein Verzeichnis-Deskriptor ist *kein* Datei-Deskriptor! `<>` kann nicht auf `Directoryhandle` angewendet werden.

Arbeiten mit Verzeichnissen

```
my $homedir = "/usr/home/drue";
opendir(DIRHANDLE, $homedir)
    or die "Couldn't open $homedir: $!\n";
while ( defined( $file = readdir(DIRHANDLE) ) ) {
    my $type = '?';
    if (-d "$homedir/$file") { $type = 'd'; }
    elif (-l -) { $type = 'l'; }
    elif (-f -) { $type = 'f'; }
    print telldir(DIRHANDLE)."$homedir/$file ($type)\n";
    # call stat("$homedir/$filename")
}
closedir(DIRHANDLE);
```

Reguläre Ausdrücke

- Was sind reguläre Ausdrücke
- Grundregeln zum Einsatz von regulären Ausdrücken
- Suchen und Ersetzen von Text
- Zeichenweises Ersetzen von Text

Suchen und Ersetzen mit regulären Ausdrücken

- Reguläre Ausdrücke sind eines der wichtigsten Werkzeuge von Perl
- Dienen zum Auffinden und Ändern von Zeichenketten in Datenmengen
- Erweitern die Funktionalität der `index`-Funktion durch mächtige Syntax
- Neben den “normalen” Zeichen können auch Escape-Sequenzen zur Definition des Suchmusters eingesetzt werden

- Zeichen mit Spezial-Bedeutung müssen mit `\` maskiert werden:

`/ . + * ? ^ $ | () [] { }`

- Text-Muster wird in Schrägstrichen geschrieben: `/Test/`

```
while (my $zeile = <FILE>) {  
    if ($zeile !~ /^#/) {  
        print $zeile;    # gib keine Kommentare aus  
    }  
}
```

Beispiele für reguläre Ausdrücke

```
/hallo/      # suche "hallo"  
/^hallo/    # ... nur am Zeilenanfang  
/hallo$/    # ... nur am Zeilenende  
/[a-z]/     # alle Kleinbuchstaben  
/[^a-z]/    # keine Kleinbuchstaben  
/[a-z]*/    # beliebig viele (0 .. x) Kleinbuchstaben  
/[a-z]+/    # mindestens ein (1 .. x) Kleinbuchstaben  
/[a-z]?/    # kein oder ein (0 .. 1) Kleinbuchstaben  
/a{2,5}/    # Folge von 2 bis 5 "a"  
/[Ff].nk/   # Klein-/Grossgeschrieben, beliebiges 2. Zeichen  
/$text/     # Sucht mit Inhalt von $text als Suchmuster
```

Escape-Zeichen zur Beschreibung von Zeichenklassen

- `\w` passt auf alphanumerische Zeichen
- `\W` passt auf nicht-alphanumerische Zeichen
- `\s` passt auf Whitespace
- `\S` passt auf alle Zeichen ausser Whitespace
- `\d` passt auf numerische Zeichen
- `\D` passt auf nicht-numerische Zeichen
- `\b` passt auf Wortgrenzen
- `\B` passt auf nicht-Wortgrenzen
- `\A` passt auf String-Anfang
- `\Z` passt auf String-Ende

Ein möglicher regulärer Ausdruck ist `/^\s+\b\w{4,8}\d*\B/`

Nutzung von Suchmustern

- Zur Anwendung der Suchmuster werden Match-Operatoren genutzt
- Match-Operatoren haben boolesches Ergebnis

`=~` Prüft, ob regulärer Ausdruck in Variable gefunden wird.
Ergebnis ist `true` bei erfolgreicher Suche

`!~` Prüft, ob regulärer Ausdruck in Variable nicht gefunden wird.
Ergebnis ist `true` bei erfolgloser Suche

```
my $name=" NetBSD" ;
if ($name =~ /BSD/i) {
    print "Bravo!\n" ;
    if ($name !~ /^net/i) {
        print "Goto http://www.netbsd.org/\n" ;
    }
}
```

Definition von Alternativen und Gruppen

- Alternativen der Suchmuster werden durch | markiert.
- Mit Klammern () werden Gruppierungen beschrieben, auf die dann bei Auswertungen und Ersetzungen zurück gegriffen werden kann
- Die Gruppen werden im Suchmuster mit \1 bis \9 referenziert

```
if ($browser =~ /Netscape|Mozilla/) {  
    print 'Gute Wahl!';  
}  
  
if ($name =~ /(Dirk)\s+(Ruediger)|\2\s+\1/) {  
    print 'Willkommen Dirk!';  
}
```

Definition von Alternativen und Gruppen

- Gruppierungen im Suchmuster sind nach erfolgreicher Suche im Ersetzungsstring und Programm über die Variablen \$1 bis \$9 ausgewertet werden.

```
$name = " Ruediger , _Dirk" ;
```

```
$name =~ s/( Ruediger ) , \s+(Dirk) / $2 $1 / ;
```

```
$vname = $2 ; $nname = $1 ;
```

Zusammenfassung: Ein paar Regeln

- Suche erfolgt von Links nach Rechts
- Variablen werden interpoliert
- Nach einem Treffer wird bei dem nächsten Zeichen fortgesetzt
- Bei Alternativen wird deren Auswertung nach der ersten zutreffenden Alternative abgebrochen
- Alle Quantoren arbeiten “greedy”
- Auf gruppierte Ausdrücke kann mit `\1 ... \9` bzw. `$1 ... $9` zugegriffen werden
- Bei der Suche wird nach Groß- und Kleinschreibung unterschieden

Ersetzen von Textteilen in Zeichenketten

- Wenn ein im Suchmuster beschriebener Text gefunden wird, kann er entsprechend dem Ersetzungsstring substituiert werden

`$name =~ s/Suchmuster/Ersetzung/option ;`

- `lvalue` wird bei Substitution modifiziert
- Nur erstes Vorkommen des Suchmusters wird ersetzt
- Ersetzung kann auch bei Wertzuweisung an Variable erfolgen

`($name = $myname) =~ s/Suchmuster/Ersetzung/option ;`

- Substitutionen arbeiten zeilenweise

Ersetzen von Textteilen in Zeichenketten

- Substitution kann durch Optionen beeinflusst werden

Option	Bedeutung
<code>s///g</code>	Ersetze alle Vorkommen des Suchmusters
<code>s///i</code>	Keine Unterscheidung zwischen Groß- und Kleinschreibung im Suchmuster
<code>s///o</code>	Suchmuster wird nur einmal im Programmlauf kompiliert
<code>s///m</code>	Zeilenumbrüche im Text beachten
<code>s///s</code>	Ignoriere Zeilenumbrüche im Text
<code>s///e</code>	Evaluieren Ersetzungsstring
<code>s///ee</code>	evaluiere Ersetzungsstring, Fehler in \$@
<code>s///x</code>	Kommentare und Whitespace in Suchmuster und Ersetzungsstring zulassen

- Optionen können beliebig kombiniert werden
- Reihenfolge der Optionen ist nicht signifikant

```
( $footer = $template ) =~ s/!DATE!/localtime/ie;
```

Ersetzen von Textteilen in Zeichenketten

- Wenn / häufig im Suchmuster benötigt wird, kann auch ein beliebiges anderes Zeichen verwendet werden

```
$usage =~ s;/usr/bin/perl;$pathtoperl;
```

```
$header =~ s|$lastdate|$today|g;
```

```
$footer =~ s{%LASTPAGE%}  
          {$lastpage}i;
```

- Substitution liefert die Anzahl der Ersetzungen als Ergebnis

```
my $count = ($text =~ s;!AUTHOR!;$author;i);
```

Look-ahead + Look-behind

- Mit Look-Ahead- und Look-Behind-Mustern kann Bedingung vor/nach dem Suchmuster definiert werden
- Look-Ahead ermöglicht Vorausschau auf das dem aktuellen Muster folgende Muster
- Look-Behind ermöglicht Rückblick auf Muster vor aktuellen Muster
- Gruppierung hat Länge=0

Ausdruck	Annahme	Beispiel
<code>(?=pattern)</code>	Positives Look-Ahead	<code>/\w+(?=\t)/</code>
<code>(?!pattern)</code>	Negatives Look-Ahead	<code>/foo(?!bar)/</code>
<code>(?<=pattern)</code>	Positives Look-Behind	<code>/(?<=\t)\w+/</code>
<code>(?<!pattern)</code>	Negatives Look-Behind	<code>/(?<!bar)foo/</code>

greedy vs. non-greedy

- *greedy = gierig*
- Ein Suchmuster ist “greedy”, d.h. es versucht so oft wie möglich zuzutreffen, solange der Rest des Suchmusters erfüllbar bleibt.
- Wenn nur der mindest-mögliche Treffer gesucht ist, muss an das Suchmuster ein ? angehängt werden.
- Damit ändert sich nicht die bedeutung, nur die “Gier” des Suchmusters

`$re1 = "baaaar" ;`

`($re2 = $re1) =~ s/ba+/c /; # cr`

`($re3 = $re1) =~ s/ba+?/c /; # caaar`

greedy vs. non-greedy

- Ein Test:

```
$re1 = "baaaar" ;
($re2 = $re1) =~ s/ba+/c/ ;
($re3 = $re1) =~ s/ba+?/c/ ;
($re4 = $re1) =~ s/ba*/c/ ;
($re5 = $re1) =~ s/ba*?/c/ ;
($re6 = $re1) =~ s/ba{2,3}/c/ ;
($re7 = $re1) =~ s/ba{2,3}?/c/ ;
print "\ $re1=$re1 , \ $re2=$re2 , \ $re3=$re3 , \ $re4=$re4 , " .
      " \ $re5=$re5 , \ $re6=$re6 , \ $re7=$re7 \n" ;
# $re1=baaaar , $re2=cr , $re3=caaar , $re4=cr ,
# $re5=caaaar , $re6=car , $re7=caar
```

Einsetzen von Kommentaren

- Kommentare helfen, komplexe reguläre Ausdrücke verständlich und pflegbar zu machen
- Angeben von Kommentaren
 - außerhalb des Musters nach #
 - mit der Option `s///x` in Suchmuster und Ersetzungsstring
 - mit dem Ausdruck `(?#Kommentar)` im Suchmuster
- Mit Option `s///x` wird
 - Whitespace ignoriert (außer in Zeichen-Klassen)
 - Text nach # wird als Kommentar betrachtet (der Literal # muss dann maskiert werden)

Einsetzen von Kommentaren

```
my $url=" http://www.mydomain.org:88/doc/regex.html#optx?q=1" ;
if ( $url =~ m{
    ( http | https | ftp | ldap | nntp | pop | imap ) : // # Schema
    ( [ . \ w - ] + ) # Hostname
    ( ? : : ( \ d + ) ) ? # Port
    ( / [ ^ # ? ] * ) ? # Path
    ( ? : \ # ( \ w + ) ) ? # Target
    ( ? : \ ? ( \ S + ) ) ? # Querystring
}x)
{ print " schema=$1 _ host=$2 _ port=$3 _ " .
    " path=$4 _ target=$5 _ querystring=$6 \ n " ; }
```

- Hinweis: Bei Gruppierung mit (?: ...) wird für diese Gruppierung keine Backreference angelegt.
- Viel einfacher geht das mit Regexp::Common

Zeichenersetzung

- Suchmuster wird zeichenweise abgearbeitet und ggf. Ersetzung durchgeführt

```
$text =~ tr /Musterzeichen/ Ersatzzeichen /Option ;
```

- Optionen können Übersetzung beeinflussen

Option	Bedeutung
tr///c	Alle außer die angegebenen Zeichen ersetzen
tr///d	Alle angegebenen Zeichen werden gelöscht
tr///s	Zusammenhängende Gruppen der Musterzeichen werden durch ein Ersatzzeichen ersetzt

- Gibt es mehr Muster- als Ersatzzeichen, so wird das letzte Ersatzzeichen angewendet

Zeichenersetzung

```
$text =~ tr/0-9/x/;      # Alle Ziffern zu 'x'  
$text =~ tr/A-Z/ /s;    # Zusammenhaengende Grossbuchstaben  
                        # zu einem Leerzeichen  
$text =~ tr/0-9/ /c;    # Alle Zeichen ausser Ziffern  
                        # zu Leerzeichen  
($t1 = $t2) =~ tr/[A-Z]/[a-z]/; # GROSS -> klein
```

Übung

Packages und Module

- Packages
- Module

Packages

- Mechanismus zur Definition von Sichtbarkeitsbereichen
- Sichtweite einer Package-Deklaration reicht von der Deklaration selbst bis zum Ende des einschließenden Blocks oder bis zur nächsten Package-Deklaration
- In den Packages definierte Variablen und Funktionen haben Gültigkeit nur in dem aktuellen Package , d.h. alle Variablen-Deklarationen innerhalb eines Packages werden in der Symboltabelle des Packages aufgenommen
- Das initiale aktuelle Package ist `main`, es ist implizit definiert
- Ein Programm kann beliebig viele Package-Deklaration enthalten
- Package-Bezeichner beginnen mit Großbuchstaben (per Konvention)

Packages

- Eine Package-Deklaration kann an einer beliebigen Stelle eingefügt werden, an der eine Anweisung stehen darf
- Auf (öffentliche) Variablen in den Packages kann durch vorangestellten Package-Bezeichner `$package::var` zugegriffen werden:
- Variablen-Bezeichner müssen (nur) innerhalb eines Packages eindeutig sein
- Auf lokale Variablen^a eines Packages kann nicht von außerhalb des Packages zugegriffen werden
- Variablen ohne Package-Bezeichner gehören automatisch zum Package `main`

^amit `my` oder `local` deklariert

Packages

```
package Oder;
$var =5;
package So;
$var =23;
package main;
$var =" Hallo" ;
print " main: _\ $var=$var\n" ;           # Hallo
print " main: _\ $Oder:: var=$Oder:: var\n" ; # 5
$Oder:: var=7;
print " main: _\ $Oder:: var=$Oder:: var\n" ; # 7
print " main: _\ $So:: var=$So:: var\n" ;   # 23
```

Module

- Module sind wiederverwendbare Packages
- Das Package wird in einer Bibliotheks-Datei mit dem selben Namen wie das Package und der Endung `.pm` abgelegt
- Thematisch zusammengehörige Module können strukturiert abgelegt werden
`Text::RTF`, `Text::HTML`, `Text::Plain::ASCII`, `Text::Plain::UTF8`
- Die Modul-Strukturen werden auf Verzeichnisse im Dateisystem abgebildet
`Text/RTF.pm`, `Text/HTML.pm`, `Text/Plain/ASCII.pm`, `Text/Plain/UTF8.pm`
- Module für viele Anwendungsfälle können beim Comprehensive Perl Archive Network (CPAN, siehe <http://www.cpan.org>) gefunden werden

Module

- Ein Programm fordert Module mit der Funktion `use` an

```
use Text::plain::ASCII;
```
- Module werden vom Perl-Interpreter beim Aufruf des Hauptprogramms hinzu geladen. Dazu durchsucht Perl die in `@INC` angegebenen Verzeichnisse:

```
# perl -e 'print "@INC\n";'  
/usr/pkg/lib/perl5/site_perl/5.6.1/i386-netbsd  
/usr/pkg/lib/perl5/site_perl/5.6.1  
/usr/pkg/lib/perl5/site_perl  
/usr/pkg/lib/perl5/5.6.1/i386-netbsd  
/usr/pkg/lib/perl5/5.6.1 .
```

Ein Modul vom CPAN installieren

- Einfachen Installieren von Modulen mit cpan

```
$ cpan Color::Rgb
```

```
# <...>
```

```
Fetching with LWP:
```

```
ftp://cpan.noris.de/pub/CPAN/authors/id/S/SH/...
```

```
# <...>
```

```
CPAN.pm: Going to build S/SH/SHERZODR/Color-Rgb-1.4.tar.gz
```

```
# <...>
```

```
Running make test
```

```
1..14
```

```
# <...>
```

```
Running make install
```

```
# <...>
```

```
/usr/bin/make install -- OK
```

Ein Modul vom CPAN installieren

- Interaktive CPAN-Sitzungen sind möglich
 - Suchen nach Autoren, Schlagworten und Modulen
 - Inkrementelles Downloaden, Bauen, Testen und Installieren der Module

- Aufruf mit dem Kommando

`cpan`

oder

`perl -MCPAN -e shell`

Perl Tipps

- Pragmas – Anweisungen für den perl-Compiler
- Fehlersuche in Perl
- Hilfe im Internet
- Buch-Tipps

Das Pragma “strict”

- Pragmas erlauben es dem Programmierer, Hinweise an den Perl-Compiler im Programmcode zu platzieren
- Mit `strict` können unsichere Sprach-Konstrukte eingeschränkt werden

```
use strict; # Anwenden aller moeglichen Restriktionen
```
- Die Restriktionen haben Gültigkeit bis zum Ende des einschließenden Blocks oder bis zur aufhebenden Anweisung

```
no strict; # Aufheben aller definierten Restriktionen
```
- Verstöße gegen die Restriktionen werden mit Compiler-Fehlermeldungen quittiert

Das Pragma “strict”

- Drei Kategorien werden unterschieden:
 - `subs` beschränkt unsicheren Gebrauch von Funktionen. Ein Compiler-Fehler wird ausgegeben beim Gebrauch eines Wortes ohne Namensraum-Bezeichner (Bareword), welches nicht als Funktion identifiziert werden kann
 - `vars` beschränkt unsicheren Gebrauch von Variablen. Ein Compiler-Fehler wird ausgegeben, wenn eine Variable ohne vorherige Deklaration gebraucht wird
 - `refs` beschränkt unsicheren Gebrauch von Referenzen. Ein Compiler-Fehler wird ausgegeben, wenn symbolische Referenzen benutzt werden

Das Pragma “strict”

- Einzelne Kategorien können selektiv aktiviert

```
# Restriktionen fuer Variablen aktivieren  
use strict 'vars';
```

oder deaktiviert

```
# Restriktionen fuer Referenzen nicht aktivieren  
use strict;  
no strict 'refs';
```

werden

- Deklaration von von Variablen erfolgt mit der Anweisung

```
use vars qw($maus $katze);
```

Das Pragma "strict"

Ein Beispiel:

```
use strict 'vars';
use vars qw/$maus @kaese/; # Deklariert $maus und @kaese
                             # im aktuellen Package
$maus = 'klein';           # O.K., global deklariert
@kaese = qq/Edam Brie/;    #           via Pragma
my $falle = 'offen';      # O.K., lokal deklariert via "my"
$ergebnis = 'leer';       # Compiler-Fehler: $ergebnis
                             #           nicht deklariert!
```

Fehlersuche in Perl

- Aufruf von Perl mit Warnmeldungen

```
perl -w perlskript.pl
```

oder Angabe im She-Bang (erste Programmzeile – nur bei Unix)

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

- Aufruf des Perl-Debuggers (siehe Hilfe-Seite `perldebug`)

```
perl -d perlskript.pl
```

- Einbauen von Fehlermeldungen im Programm

```
print STDERR "$0: _Probleme!" if $PROBLEM;
```

Sie werden auf der Standard-Fehlerausgabe ausgegeben und können so leicht von der eigentlichen Programm-Ausgabe getrennt werden.

- Nutzen von Logging-Frameworks

Fehlersuche in Perl

- Prüfen, ob das Skript die notwendigen Berechtigungen besitzt
 - Unix: `chmod 0755 perlskript.pl`
 - Win32: `PATHEXT=.pl;.com;.exe;.bat`

- Ist das Script Standard-Perl

```
perl -wc perlskript.pl
#      ^ nur kompilieren
```

- Sind die eingebunden Module in der erforderlichen Version vorhanden

```
perl -MText::HTML -e 'print Text::HTML->VERSION'
2.40
```

Hilfe zu Perl

- Perl besitzt ein umfangreiches Hilfe-System
- Einzelne Sektionen beschreiben Sachverhalte zu Perl, z.B.
 - perl – Allgemeine Informationen
 - perlsyn – Syntax von perl, Kontrollstrukturen, Schleifen
 - perlop – Alles über Operatoren
 - perlfunc – Perl's eingebaute Funktionen
 - perldata – die verschiedene Datentypen
 - perlsub – Arbeiten mit benutzerdefinierten Funktionen
 - perlre – reguläre Ausdrücke (Perl's großer Trumpf ;-)
 - perllocal – lokal installierte Packages

Hilfe zu Perl

- Aufruf der Hilfe an der Kommandozeile

```
perldoc perlsektion
```

- und außerdem: Aufruf der Hilfe-Seiten unter Unix

```
man perlsektion
```

- oder: Aufruf unter Windows (ActivePerl) durch Browsen der HTML-Seiten im Unterverzeichnis `html` der Perl-Installation

- Aufruf der Hilfe zu einer Perl-Funktion

```
perldoc -f function
```

Hilfe im Internet

- <http://www.perl.com> – die Zentrale
- <http://www.perl.org> – Sammlung von Links
- <http://www.cpan.org> – Software-Archiv
- <http://www.perl.de> – die deutschsprachige Perl-Community
- news:comp.lang.perl.* – News-Foren

Lesestoff

- Damian Conway: *Perl Best Practices*. O'Reilly, 1. Auflage, 2005.
- Larry Wall, Tom Christiansen, Jon Orwant: *Programming Perl*. 3. Auflage, O'Reilly, 2000.
- Johan Vromans: *Perl 5 kurz und gut*. O'Reilly, 4. Auflage, 2003.
- Tom Christiansen, Nathan Torkington: *Perl Cookbook*. O'Reilly, 2. Auflage, 2003.
- The Perl Journal, <http://www.tpj.com/>

Und noch was . . .

Die aktuelle Version dieses Seminarfoliensatzes ist im Internet zu finden:

<http://niebegeg.net/>

Ich wünsche Euch viel Spaß mit Perl!