Scriptorientierte Programmiertechnik – Perl

Prof. Dr.-Ing. Torsten Finke

FOM

16. August 2011(Rev.: bb133941e038)

Überblick

Klausur

Literatur

Arbeitsumgebung

Perl Programmierung

Mehr Scriptprogrammierung

Klausur – Formalia

- Dauer 120 Minuten
- keine Hilfsmittel
- Formvorschriften
  - verbindliche Hinweise beachten
  - leserlich schreiben
  - Korrekturrand beachten
  - nur Vorderseiten beschreiben
  - nicht mehr als **eine** Aufgabe pro Blatt bearbeiten

Klausur – Inhalte

- Inhalt komplett relevant
- kein Repetitorium
- Auswahlklausur
- Schwerpunkt auf Verständnis

Literatur

- Schwartz, Randal L., Tom Phoenix: *Learning Perl*. O'Reilly & Associates, 2008, ISBN: 978-0596520106.[a]
- Wall, Larry, Tom Christiansen, Jon Orwant: *Programming Perl*. O'Reilly, 2000.
- Christiansen, Tom, Nathan Torkington: *Perl Kochbuch. Beispiele und Lösungen für Perl-Programmierer*. O'Reilly & Associates, 2004, ISBN: 3897213664.
- Conway, Damian: *Object Oriented Perl*. Manning Publications.
- Conway, Damian: *Perl Best Practices*. O'Reilly, 2005, ISBN: 0596001738.

[a] diese Foliensammlung orientiert sich stark an *Learning Perl*

Editoren

- Emacs, Vim
- Notepad++
- Eclipse, EPIC

Arbeitsumgebung

- es gibt nur einen Weg, Programmieren zu lernen: Programmieren!
- beschaffen und installieren Sie sich eine Arbeitsumgebung!
- vollziehen Sie die Programmierbeispiele nach!
- absolvieren Sie die Übungen regelmäßig!

Perl-Interpreter

- auf vielen Plattformen verfügbar
- unter Unix meist vorhanden
- Windows: `http://www.activestate.com/Products/activeperl/index.mhtml`
- Distribution via `http://www.cpan.org`
- integrierte Online-Dokumentation via `perldoc`

Was ist Perl?

- Practical Extraction and Report Language
- Pathologically Eclectic Rubbish Lister

```
#!/usr/bin/perl
foreach ( `perldoc -u -f atan2` ) {
    s/\w<([^>]+)>/\U$1/g;
    print;
}
```

- Interpreter (Just-in-Time compiler)
- Schwach typisiert
- Prozedural/Objektorientiert
- Sprachmächtig, viele Zusatzmodule
- Online Dokumentation *perldoc*
- Autor: Larry Wall und viele weitere Contributoren

Wofür Perl?

- Einsatzgebiete:
  - Arbeiten auf Textdaten (sehr stark)
  - WWW (CGI)
  - Systemadministration
  - Kleine bis mittlere Softwareprojekte (Beispiel BackupPC)
  - Command line tools
  - GUI (mit Tk oder Qt)
- Wo eher nicht?
  - Hardwarenahe Software
  - Nichtoffener Quelltext

Hello World!

- Datei: `hello.pl`
- Windows:
  - Suffix `.pl` mit Ausführung des Interpreters `perl` verknüpft
  - Aufruf in Eingabeaufforderung: `.\hello.pl` oder `perl hello.pl`
- Un*x:
  - `chmod +x hello.pl` deklariert Datei als Programm
  - Aufruf aus Terminal: `./hello.pl`

```
#!/usr/bin/perl

print "Hello World!\n";
```


Übungen

- Provozieren Sie Fehler:
  - Dateisuffix verändern
  - Ausführbarkeit der Datei unterbinden (unter Linux)
  - erste Zeile des Programms manipulieren (was bedeuten die Teile der ersten Zeile?)
  - Semikolon löschen
  - Anführungszeichen löschen
- Verändern Sie die Wirkung des Programms
  - `\n` entfernen
  - Ausgabetext verändern

Scalare Daten

- elementarer Datentyp
- Komponente aggregierter Datentypen
- schwach typisiert
- meist Zahlen oder Zeichenketten

```
#!/usr/bin/perl
print ...
```


Zahlen

- Zahlen intern als `double` gespeichert(C, meist IEEE 754)
- Ganzzahlarithmetik kann erzwungen werden:
  - Pragma `integer`
  - Wirkung auf arithmetische Operationen beschränkt, Bespiel $3/2 \to 1$
  - interne Speicherung als `int`
- wo sinnvoll, wird automatisch in Ganzzahlen gewandelt
- unterscheide Zahl und ihre Darstellung!
- Wandlung Zahl ↔ Zeichenkette bei Bedarf
- siehe *perldoc*: `perldata`

Fließpunkt-Literale

- `1.`
- `1.5`
- `0.2`
- `.5`
- `+3.14`
- `-2.718`
- `6.023e23` $= 6.023 \times 10^{23} = 602300000000000000000000$
- `1.05457e-34` $= 1.05457 \times 10^{-34}$
- `1E3` $= 1000$

Ganzzahl-Literale – Integers

- dezimal
  - 0
  - 1
  - 42
  - -2001
  - 1680801063338
  - `1_680_801_063_338` = 1680801063338
- nondezimal
  - oktal: führende Null
    `0377` = 255
  - hexadezimal: führend `0x`
    `0xff` = `0xFF` = 255
  - binär: führend `0b`
    `0b11111111` = 255
  - `0x4A_7B_D3_EC`

Arithmetische Operatoren

- `2 + 3`
- `2.3 + 1.34E1`
- `3 * 4`
- `15 / 3`
- `5 / 3`
- `1.2 / 0.3`
- `13 % 5` (Modulo, Divisionsrest; Ganzzahlrechnung)
- `2 ** 3` (Potenz)
- siehe *perldoc*: `perlop`

Bit-Operatoren

- `13 & 11`
- `13 | 11`
- `13 ^ 11`
- `7 << 3`
- `0xff >> 4`
- `~13`

Zeichenketten – Strings

- String: Folge von Zeichen (z.B. Bytes)
- Kodierung systemabhängig, z.B. ASCII
- kürzester String: leer
- längster String: entsprechend verfügbarem Speicher
- Zeichenfolgen beliebig interpretierbar (Bilder, Sound usw.)

Zeichenketten-Literale – Single Quoting

- `'hello'`
- `''`
- `'Apostroph \'im Text'`
- ```
  'hello
  world'
  ```
- `'gute\n Tag'`
- `'\'\\'`
  ```

Zeichenketten-Literale – Double Quoting

- `"hello"`
- `"hello␣world\n"`
- `"Zitat:␣\"Hello␣World\""`
- `"Kurzzitat:␣'hello'"`
- `"hello\tworld"`

Zeichenketten – Sonderzeichen

`\n`, `\r`	line feed, carriage return
`\t`, `\f`	tab, form feed
`\a`, `\e`	beep, escape
`\007`, `\x7f`	octal, hexadecimal codes
`\l`, `\u`	next letter lower (upper) case
`\L`, `\U`, `\E`	following characters lower (upper) case until `\E`

Zeichenketten-Operatoren

- `"hello" . "world"` – Verkettung
- `"hello" . '␣' . "world"`
- `"hello" x 3` – Repetition
- `"hello" x (3 + 2)`
- `"hello" x (3 / 2)`
- `3 x 'hello'`
- `3 x 4`

Konversion: Strings ↔ Zahlen

- Kontext:
  - Operatoren definieren den Kontext der Operanden
  - Operanden werden bei Bedarf gewandelt
  - Semantik der Operatoren bleibt erhalten
- `3 + 4`
- `3 . 4`
- `3 + 'hello'`
- `3 + '4hello'`
- `3 + '4.3hello'`
- `3 + '042hello'` – dezimal
- *`#!/usr/bin/perl -w    # Warnung`*

Vergleichsoperatoren

Vergleich	arithmetisch	textlich
gleich	==	eq
ungleich	!=	ne
kleiner	<	lt
kleiner oder gleich	<=	le
größer	>	gt
größer oder gleich	>=	ge
Reihenfolge	<=>	cmp

- `03 == 3`
- `42.0 < 7 x 7`
- `"123" eq '123'`
- `'␣' gt ''`
- `"abc" cmp 'ABC'`

Weitere Scalare Daten

- `undef`
- Referenzen
- File-Handles

Scalare Variable

- `$variable_mit_langem_namen`
- `$a`
- `$_auch_eineVariable`
- `$ABC` – keine Großbuchstaben am Namensanfang!
- sprechende Namen
- Autovivifikation
- siehe *perldoc*: `perlvar`

Scalare Zuweisung

- `$x = 42`
- `$name = "Alfred"`
- `$sum = 3 + $name`
- `$lied = 'tra' . 'la' x 3`
- `$x = $x + 4`
- `$x += 4`
- `$a .= "hello"`
- `$x **= 3`
- `$z = $y = $x += 3`
- `$x = (2,3 + 5)`

Interpolation

- `$name = "Alfred"`
- `print "Hallo␣$name"`
- `print 'Hallo␣$name'`
- `print "x␣=␣$x\n"`
- `print 'x␣=␣' . $x, "\n"`

Operatoren

- Arithmetik
- Zeichenketten
- Zuweisung
- Vergleich
- Kardinalität (unär, binär, ternär)

Operatoren – Assoziativität und Präzedenz

links	->
nichtassoziativ	++ --
rechts	**
rechts	! ~ \ *unär:* + -
links	=~ !~
links	* / % x
links	+ - .
links	<< >>
nichtassoziativ	< > <= >= lt gt le ge
nichtassoziativ	== != <=> eq ne cmp ~~
links	&
links	| ^
links	&&
links	|| //
nichtassoziativ	.. ...
rechts	?:
rechts	= += -= *= etc.
links	, =>
rechts	not
links	and
links	or xor

Übungen

Was ergeben die folgenden Statements?

- `$x = 039`
- `2 + 3 * 4`
- `9 - 4 - 2`
- `2 ** 3 ** 4`
- `5.7 % 2.1`
- `~0`
- `'hello' x 'world'`
- `3 * 4 x 5`
- `3 x 4 * 5`
- `3 + 4 . 5`
- `$y = 2; $x = $y += 3`
- `$x = (3,4)`
- `$y = 4; $x = ( $y < 5 )`

Fallunterscheidung – Selektion

```
$x = 3;
if ( $x % 2 == 1 ) {
    print "$x␣ist␣ungerade\n";
}

$x = 3;
if ( $x % 2 == 0 ) {
    print "$x␣ist␣gerade\n";
} else {
    print "$x␣ist␣ungerade\n";
    print "$x␣ist␣eventuell␣prim";
}
```


Was ist wahr?

```
$x = 42;
if ( $x ) { print "$x␣ist␣wahr\n" }
```

- falsch:
  - 0
  - ''
  - '0'
  - undef
- wahr:
  - alles andere
  - 1
  - "abc"
  - "␣␣␣"
  - '00'
  - 'false'

Eingabe

```
$x = <STDIN>;
chomp $x; # remove linebreak
if ( $x eq $x + 0 ) {
   print "Eingabe␣$x␣numerisch␣interpretierbar\n"
}
```

Achtung: was bewirkt `$y = chomp($x)`

Schleife – Iteration

```
$n = 5;
while ( $n ) {
    "n␣=␣$n\n";
    $n -= 1;
}

$x = undef; # clear $x
while ( $x ) { print "hello\n" }

$x = <STDIN>;
if ( defined $x ) {print "Eingabe␣war␣$x"}

while ( $x = <STDIN> ) {
    print $x
}
```

Kommentare

- # bis Zeilenende
- Autor
- Datum
- Zweck
- unsinnige Kommentare `$x += 1; # x inkrementieren`

Liste – Feld

- aggregierter Datentyp
- numerischer Index (ab Null)
- Feld, Array: benannte Liste, Listenvariable
- Liste: unbenannte Liste
- Aufbau aus Scalaren.

Feldelemente

- Zugriff:

```
$a[0] = "hello";
$a[1] = "world";
$a[2] = 42;
$a[5] = "hop";
```

- Nutzung:

```
$a[2] / 7;
$a[1] x 2;
```

- Was ist `$a`?

Spezielle Feldindizes

- Ende: `$n = $#a`
- `$a[$#a]`
- `$a[-1]`
- `$a[-100]`
- `$a[100]`

Listen-Literale

- `(1, 2, 3)`
- `(1, 2, 3,)`
- `("hello", "world", 42)`
- `(1 .. 13)`
- `(7.3 .. 12.5)`
- `(9 .. 0)`
- `(0 .. $#a)`
- `($x + $y, $a . $b)`
- `qw/ hello world /`
- `qw( hello world )`

Listenzuweisung

- `($a, $b, $c) = ("hello", "world", 42)`
- `($a[0], $a[1], $a[2]) = ("hello", "world", 42)`
- `@a = ("hello", "world", 42)`
- `@gruss = qw ( hello world )`
- `@intervall = (0 .. 1e3)`
- `$a = 13; @z = (@a, undef, @a, $a)`
- `@a = $a`
- `@b = @a`
- `@a = ()`
- `$#a = -1`
- `@a = undef # Oops!`

Feldoperatoren

- `@a = ("hello", "world", 42)`
- `$last = pop @a`
- `pop @a`
- `push @a, "13"`
- `push @a, (1 .. 9)`
- `$first = shift @a`
- `unshift @a, "123"`
- `push @a, shift @a`
- `pop`, `shift` auf leerem Array: `undef`

Listeninterpolation

- `@text = qw( Perl Prgramm )`
- `print "Dies␣ist␣ein␣@text\n"`
- `print "Dies␣ist␣ein␣@{text}lein\n"`
- `print "Dies␣ist␣ein␣$text[-1]lein\n"`
- `$n = "3␣-␣2"`
- `print "Dies␣ist␣ein␣$text[$n]lein\n"`
- `print @text`

Listeniteration

- `@languages = qw(C Java Perl Fortran Assembler)`
- `foreach $lang ( @languages ) { print "$lang\n" }`
- `foreach ( @languages ) { print }`
- `foreach ( @languages ) { print "$_\n";  }`

Operationen auf Listen

- `@a = (1 .. 12)`
- `@r = reverse @a`
- `@a = reverse @a`
- `@s = sort @a`
- `@s = reverse sort @a`
- `@s = sort reverse @a`
- `@s = sort { $a <=> $b } @a`
- `@m = map { 3 * $_ } @a`
- `map { $_ *= 3 } @a`
- `@g = grep { $_ % 2 == 0 } @a`

Listenkontext

- `@a = (1 .. 12); @b = qw( hello world )`
- `@a = (2, 3, 5)`, aber `$a = (2, 3, 5)`
- `3 + @a`
- `$n = @a`
- `($n) = @a`
- `@z = reverse @a`
- `$z = reverse @a`
- `$z = reverse "foobar"`
- `@t = 3 + 4`
- `print "Intervall␣hat␣", scalar @a, "␣Werte\n"`
- `print "Intervall␣hat␣", @a + 0, "␣Werte\n"`

Feldbereiche

- `@a = (1 .. 9)`
- `$x = $a[3]`
- `@m = @a[2..7]`
- `@n = ( 2 .. 7 ); @m = @a[@n]`
- `splice @a, 2, 3, qw/foo bar/`
- `@rem = splice @a, 2, 3`
- `$rem = splice @a, 2, 3`

Listeneingabe

- `@text = <STDIN>`
- `chomp @text`
- `chomp(@text = <STDIN>)`

Übungen

- Was bewirkt die folgende Anweisung? `($x, $y) = ($y, $x)`
- Auf welche Weise kann ein Array gelöscht werden?
- Schreiben Sie ein Programm, das die Anzahl der eingelesenen Zeilen ausgibt!
- Schreiben Sie ein Programm, das die eingelesenen Zeilen in umgekehrter Reihenfolge wieder ausgibt!
- Wie können die Funktionen `pop`, `push`, `shift` und `unshift` durch `splice` nachgebildet werden?

Funktionen

- Listenoperatoren oder unäre Operatoren
- Listenkontext der Argumente
- Klammern:
  - was aussieht wie eine Funktion, ist eine Funktion
  - Klammern dürfen fehlen
  - ohne Klammern gilt Operatorpräzedenz
- siehe *perldoc*: `perlfunc`

Numerische Funktionen

- `sqrt $x # Wurzel`
- `abs $x # Betrag`
- `exp $x # e^x`
- `log $x # Logarithmus naturalis`
- `sin $x, cos $x # Trigonometrie`
- `atan2 $y, $x # Arcus-Tangens [-Pi:Pi]`
- `$pi = 4 * atan2 1,1`
- `rand $n # Zufallszahl < n`

Zeichenkettenfunktionen

- `length $line`
- `reverse $line`
- `index $line, $word, $startpos`

```
$p = 0;
while ( ( $p = index $line, $word, $p ) >= 0  ) {
    print "found␣$word␣at␣position␣$p\n";
    $p += 1;
}
```

- `rindex $line, $word, $startpos # last`
- `$ex = substr $line, $offset, $length`
- `substr $line, $offset, $length, $repl`

Umwandlungsfunktionen

- `hex $string; oct $string # String als hex./oct. Zahl`
- `chr $num  # Zeichen der Codetabelle`
- `ord $char # Nummer des Zeichens`
- `uc $a; lc $a; ucfirst $a # groß/klein`

Benutzerdefinierte Funktionen

- Erweiterung Funktionsumfang
- Wiederverwendung von Code
- Unterschied Funktion/Subroutine
- Qualifizierer & (optional)

Unterprogramm – Definition

```
sub mehr {
    $n += 2;
    print "n␣=␣$n\n";
}
```

- Definition an beliebiger Stelle im Quelltext
- Definition global
- Mehrfachdefinition möglich

Unterprogramm – Aufruf

- `$n = 13; &mehr`
- `&mehr`

Unterprogramm – Rückgabewerte

Rückgabewert: letzter evaluierter Wert

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

sub sum_debug {
    $c = $a + $b;
    print "c␣=␣$c\n";
}

sub fun {
    if ( $a > 3 ) {
        $x;
    } else {
        ( 1 .. 5 );
    }
}
```



```
sub fun {
    if ( $a < 0 ) {
        return -1;
```


Unterprogramm – Argumente

```
sub max {
    if ( $_[0] > $_[1] ) {
        $_[0];
    } else {
        $_[1];
    }
}

$m = &max(7, 5, 11); # oops
```

- Übergabe per `@_`
- Übung: *call by reference* oder *call by value*?

Private Variable

```
sub max {
    my ($a, $b) = @_;
    if ( $a > $b ) {
        $a;
    } else {
        $b;
    }
}

$a = 5; $b = 7; print &max($a, $b);
```

Diskurs – *local*

- älteres Perl-Idiom: local
- Variable bleibt global, erhält lokale Überblendung

```
$g = "foo";

sub show {
    print $g;
}
sub local_show {
    local($g) = "bar";
    &show();
}
sub my_show {
    my($g) = "bar";
    &show();
}
```



```
&my_show();     # bar
```


Parameter-Listen

```
sub max {
    if ( @_ < 2 ) {
        print "minimum␣2␣args!";
    }
    my ($s) = shift(@_); # why?
    foreach ( @_ ) {
        if ( $_ > $s ) {
            $s = $_;
        }
    }
    $s;
}

$x = &max(7, 5, 13);
$x = &max();
```


Lokale Variable nutzen

- geschachtelte Sichtbarkeit
- Tip: grundsätzlich alle Variablen mit my lokalisieren
- überall verwendbar

```
my($s) = 0;
foreach ( 0 .. 7 ) {
    my ($d) = 2 * $_;
    $s += $d;
}
print $s;
```

- Kontext:

```
my ($a) = @_; # list
my $a   = @_; # scalar
```


Pragma – *strict*

```
$dideldum = 42;
print $dideldumm; # oops!
```

- Pragma use strict
  - reduziert Namensfehler
  - erzwingt Deklaration (per my)
- verbessert Wartbarkeit

Übergabe getrennter Arrays

- Beipiel: komponentenweise Addition zweier Arrays
- Problem: Arrays werden in `@_` verschmolzen

```
sub add {
    my ( @a, @b ) = @_; # @b empty!
    ...
}
...
add(@x, @y);
```

- Lösung: Referenzen
- siehe *perldoc*: `perlref`, `perlreftut`

Referenzieren

- Referenz enthält Verweis auf Variable (oder Funktion)
- Analogie zu Pointern in C
- Referenz entspricht Adresse (im Speicher)
- Referenzoperator \:

```
$scalarref = \$foo;
$arrayref  = \@array;
$subref    = \&tool;
```

- Referenzen sind immer Skalare

Anonyme Referenzen

- unbenannte Liste:

```
$aref = [ 42, "foo", -sqrt 2 ];
```

- Unterschied ( ) und [ ]!
- unbenanntes Unterprogramm:

```
$sref = sub { print "hello\n" }
```

- `sub` liefert Codereferenz

De-Referenzieren

- Dereferenzierung per Typqualifizierer:

```
@a = @{$arrayref};
$x = ${$scalarref};
```

- Vereinfacht (wenn Referenz atomar):

```
@a = @$arrayref;
$x = $$scalarref;
```

Referenzen anwenden

- kopieren: `$bref = $aref;`
- Listen (Referenzen sind Skalare):

```
@a = ( $aref, \$liste, [ 1, "bar" ] );
```

- Feldeintrag

```
@a = ( 42, "foo", -1 );
$aref = \@a;
print ${$aref}[1]; # prints "foo"
print $aref->[1]; # the same (using arrow operator)
```

- Feldmanipulation

```
push @$aref, "hello"; # append "hello" to @a
```


Komplexe Referenzen

- mehrfach referenzieren

```
@z = ( "foo", "bar" );
$y = \@z;
$x = \$y;
$w = \$x;
print ${${${$w}}}[0]; # prints "foo"
```

- Lists of lists (zum Beispiel Tabellen)

```
@a = ( "foo", 42, "bar" );
@r = ( \@a, [2, 3, 5], [ "hello", \@a ]);
print $r[2]->[1]->[2]; # prints "bar"
print $r[2][1][2]; # arrow optional between brackets
```


Referenz abfragen

- Liste

```
@a = ( 1, 2, 3 );
$a = \@a;
$r = ref $a; #  $r now "ARRAY"
```

- Subroutine

```
$sref = sub foo { print "hello"; };
$r = ref $sref; #  $r now "CODE"
```


Beispiel für Referenzen

```
sub add {
    my ($a, $b) = @_;
    foreach ( @$a ) {
        $_ += shift @$b;
    }
}
@x = ( 1 .. 5);
@y = ( 2, 3, 5, 7, 11);

add(\@x, \@y); # Nebenwirkung?

print "@x\n";
```

Verwendung des Ampersand

- Ampersand (&) qualifiziert Subroutines
- vermeidbar, wenn:
  - Nutzung per Rückwärtsreferenz
  - keine Namenskollision zu Builtins
- bei Aufruf mit Ampersand *müssen* Klammern gesetzt werden
- bei Aufruf ohne Ampersand dürfen Klammern fehlen

Rekursion

- Unterprogramme können rekursiv aufgerufen werden
- Speicherverbrauch und Rechenzeit beachten
- gegebenenfalls in Iteration wandeln

Übungen

- Schreiben Sie ein Unterprogramm, das die Summe einer Liste berechnet
- Schreiben Sie ein Unterprogramm, das Minimum, Mittelwert und Maximum einer Liste berechnet.
- Schreiben Sie ein Unterprogramm, das die mittlere Wortlänge einer Liste berechnet.
- Gegeben sei eine Tabelle als Liste von Zeilen. Schreiben Sie ein Unterprogramm, das eine Liste der Spalten dieser Tabelle zurückgibt.

Datenstruktur – Hash

- assoziative Liste
- (Schlüssel, Wert) – Skalare
- Aggregierter Datentyp
- allgemeine Zeichenketten als Schlüssel
- Schlüssel eindeutig
- Hash-Algorithmus

Hash – Verwendung

Schlüssel	Wert
Rechnername	IP-Adresse
Artikelnummer	Preis
Wort	Anzahl
Dateiname	übergeordnetes Verzeichnis

Hash – Zugriff auf Elemente

- `$hash{$key}` – geschweifte Klammern
- `$preis{"08X15"}`
- `$preis{"47Y11"} = 5.67`
- `$p = $preis{5 + 6 * 7 . "Y11"}`
- `$wert{76574351985478}` vs. `$wert[76574351985478]`
- `$wert{$index}` – Index bedeutungsfrei

Hash – Zuweisung

- `%hash` – Gesamthash
- `%preis = ( "CPU", 99.95, "RAM", 39.90, "HD", 139.00 )`
- `@preisliste = %preis` – Reihenfolge
- `%a = %b`
- `%a = reverse %b`

```
my %price = (
    CPU =>  99.95,
    RAM =>  39.90,
    HD  => 139.00,
);
```


Schlüssel und Werte

- `my @k = keys %hash`
- `my @v = values %hash`
- `my $c = keys %hash`
- `my $h = %hash # true/false`

Hash-Iteration

```
while ( ( $key, $val ) = each %hash ) {
    print "$key␣=>␣$val\n";
}

foreach $key ( sort keys %hash ) { # Speicherbedarf
    print "$key␣=>␣$hash{$key}\n"; # Interpolation
}
```


Hashelement – Existenz und Löschung

- `$hash{"data"} = undef`
- `if ( exists ($hash{"data"}) ) { print "exists"}`
- `delete $hash{"data"}`
- `%h = ()`

Hash – Interpolation

- `print "$hash{$key}"`
- `print "%hash"; # oops`

Hash – Teilfelder

```
%h = (
       foo   => 123,
       bar   => "hello",
       dummy => "crash" );
@k = qw /foo bar/;
@v = @h{@k};
```

Hash-Referenzen

- Übergabe von Hashes an Subroutines:
  - in Gestalt eines Arrays:

```
sub foo {
    my (%h) = @_;
    ...
}
...
foo( %hash )
```

  - effektiver als Referenz:

```
sub foo {
    my ($href) = @_;
    my %h = %{$href};
    ...
}
...
foo( \%hash )
```

- auch bei Rückgabe möglich: `return \%h;`

Anonyme Hashreferenz

- analog zur Liste:

```
$a = [ 13, 'bar', \@b ];
```

- analog zur Liste:

```
$h = { Name => 'Meyer', Vorname => 'Emil' };
```

- Typ der Referenz:

```
$r = ref $h; #  $r now "HASH"
```


Hashreferenzen für komplexe Datenstrukturen

- hashes of hashes (hoh)
- hashes of lists (hol)
- lists of hashes (loh)
- lists of lists (lol)
- beliebig komplexe Datenstrukturen

Beispiel für komplexe Datenstruktur

```
$db = [
        {
          ID => 12345,
          Name => 'Schulze',
          Fon => {
            Fix => [ '+49201918273', '+3312345678' ],
            Mobil => [ '01713377214' ],
          }
        },
        { ID => 23456, },
      ];
...
$id = $$db[0]{'ID'};
@tel = @{$$db[0]{'Fon'}{'Fix'}};
print "ID=$id,␣Tel:␣@tel\n";
```

Referenzen – Beispiel: XML-Datei

```
<computer>
  <pc name="anton">
    <network>
      <ipaddress>192.168.1.101</ipaddress>
      <nameserver>192.168.1.1</nameserver>
      <nameserver>192.168.1.254</nameserver>
    </network>
    <hardware>
      <cpu type="quadcore" freq="2.0␣GHz"/>
      <disk>
        <type>S-Ata</type>
        <size>500 GB</size>
      </disk>
    </hardware>
  </pc>
  <pc name="bert">
  </pc>
</computer>
```


Komplexe Datenstrukturen – XML

```
use XML::Simple;        # XML-Parser
use Data::Dumper;       # output complex data structures
$x = XMLin("pc.xml");   # get XML file
print Dumper($x);       # show content
$dns = $$x{'pc'}{'anton'}{'network'}{'nameserver'}[0];
```


Übungen

- Schreiben Sie ein Programm, das jede Zeile als ein Wort einliest und die sortierte Liste aller distinkten Wörter ausgibt!
- Schreiben Sie ein Programm, das jede Zeile als ein Wort einliest und für jedes Wort die Häufigkeit ausgibt!
- Wie kann vorab geprüft werden, ob ein Hash revertiert werden kann?
- Bilden Sie mit Hash-Operationen die Mengen-Operationen *Vereinigung*, *Restmenge* und *Schnittmenge* nach! Hund, Katze, Maus, Strauß und Kobra sind Landtiere. Hund, Katze, Maus, Robbe, Delfin und Fledermaus sind Säugetiere. Bestimmen Sie:
  - alle Tiere
  - Tiere, die nicht an Land leben
  - Säugetiere, die an Land leben
- Schreiben Sie Unterprogramme, die Mengen als Hashes erhalten und für zwei gegebene Menge jeweils Vereinigungsmenge, Schnittmenge und Restmenge der ersten ohne die zweite Menge bestimmen.

Zusatz-Übung – Referenz

Gegeben ist die folgende Vorgängerabhängigkeit von Programmiersprachen:

```
%vorfahr = (
    'C' => 'Fortran',
    'C++' => 'C',
    'PL/1' => 'Cobol',
    'Pascal' => 'PL/1',
    'Fortran' => 'Assembler',
    'Cobol' => 'Assembler',
    'Java' => 'C++',
    'Lisp' => 'Assembler',
    'Prolog' => 'Lisp',
    'Scheme' => 'Lisp',
);
```

Schreiben Sie ein Programm, das für alle gegebenen Programmiersprachen jeweils die komplette Vorgängerlinie ausgibt!

Zusatz-Übungen – Referenzen

- Ein typisches Dateiformat ist das INI-Format. Es ist in Abschnitte mit Titeln in eckigen Klammern gegliedert. Darin stehen Zeilen der Form *Parameter* = *Wert*. Schreiben Sie ein Unterprogramm, das INI-Dateien einliest und als Referenz auf einen HoH zurückgibt.
- Nutzen Sie das Modul `XML::Simple`, um die oben eingelesene Datei im XML-Format abzuspeichern und das Modul `Data::Dumper`, um diese Datenstruktur auszugeben.

Standard Input

- Lesen von *stdin*

```
while ( defined( $line = <STDIN> )) {
    print "$.:␣$line"; # $. - line number
}
```

- Kurzform

```
while ( <STDIN> ) {  # implizite Zuweisung an $_
    print "$.:␣$_";
}
```

- Listenkontext

```
foreach ( <STDIN> ) {
    print "$.:␣$_";
}
```


Diamond Operator <>

- Eingabe über *stdin*
  - `cat foo | myprog.pl`
  - `myprog.pl < bar`
- Eingabe aus Dateinamen als Aufrufargument
  - `myprog.pl foo.dat bar.dat`
- Idee:
  - wenn *stdin*, dann von dort lesen
  - wenn Dateinamen übergeben, dann aus Dateien lesen

```
while ( <> ) {
    print "$.:␣$_";
}
```


Aufrufargumente

- Aufrufargumente erscheinen in `@ARGV`
- `@ARGV` wie normales Feld zu behandeln (`shift`, `foreach`)
-

```
@ARGV = qw/foo bar/ # read from these files
while ( <> ) {
    print "$ARGV/$.:␣$_"; # $ARGV - filename
}
```

Standard Output

- `print "hello";`
- `print STDOUT "hello"; # no comma!`
- `print ( 2 + 3 ) * 4`
- Ausgabe gepuffert, Autoflush: `$| = 1`
  - `foreach ("a" .. "z") {print; sleep 1}`
  - `$| = 1; foreach ("a" .. "z") {print; sleep 1}`

Formatierte Ausgabe

- formatierte Ausgabe analog zu C
- `printf "hello␣%s,␣the␣answer␣is␣%d", $user, $n`
- `printf "pi/4␣=␣atan(1.0)␣=␣%g", atan2(1,1)`
- `printf "n␣=␣%6d", $n;  # right justified`
- `printf "%-20s", $name; # left justified`
- `printf "pi␣~␣%8.4f", 22/7;`
- `printf "Rabatt␣=␣%.2f%%", 3/100`
- `$fmt = "%6d" x @num; printf $fmt, @num`
- `unshift @num, "%6d" x @num; printf @num`

Interne Ausgabe

- Analog zu formatierter Ausgabe
- Erstellen von Strings zur internen Verarbeitung
- `$pi = sprintf "%.2f", 4 * atan2(1,1); # round`
- `$date = sprintf "%.4d-%.2d-%.2d", $y, $m, $d`
- Vorsicht: Listenkontext **nicht** analog zu *printf*

Übungen

- schreiben Sie ein Programm, das alle Aufrufargumente zeilenweise nummeriert ausgibt.
- Eine Datei enthält:

```
hello
world
and
friends
```

was liefert die folgende Sequenz? `@t = <>; print "@t"`

Regulärer Ausdruck – Idee

- Woran erkennt man:
  - eine Zahl
  - eine Uhrzeit
  - eine Email-Adresse
  - ein Absatzende?
- Muster!
- Nicht verwechseln mit Dateinamensmustern!

Einfache Muster – Pattern

```
$_ = "dideldum";
if ( /del/ ) {
    print "passt!";
}
```

- Muster zwischen //
- Muster passt oder passt nicht

Meta-Zeichen

- `/3\.14159/` vs. `/3.14159/`
- Punkt passt auf **ein** Zeichen
- Punkt passt auf **jedes** Zeichen (bis auf Zeilenende)
- Schutz durch \
- Aber: `\t` – Tabulator

Quantifizierer

- `/fo*bar/`
- `/fo+bar/`
- `/fo?bar/`
- `/fo{3}bar/`
- `/fo{3,5}bar/`
- `/fo{3,}bar/`

Gruppierung

- `/foo+/` vs. `/(foo)+/`
- `/((foo)+(bar)+)?/`
- `/(foobardideldum)*/` passt auf `"hello␣world"`!

Alternation

- `/foo|bar|hello/`
- `/foo( |\t)+bar/`
- `/foo( +|\t+)bar/`
- `/foo (and|or) bar/`

Muster-Test

```
#!/usr/bin/perl
while ( <> ) {
    chomp;
    if ( /MUSTER/ ) {
        print "passt:␣|$`<$&>$'|\n"
    } else {
        print "passt␣nicht\n";
    }
}
```


Greediness

- reguläre Ausdrücke normalerweise *gierig* (greedy)
- `$_ = "AxyzApqrA";`
- `/A.*A/` – greedy
- `/A.*?A/` – nongreedy
- `/A[^A]*A/` – oft ähnlich wie nongreedy
- Matching-Variable: `` $` ``, `$&`, `$'` – langsam!

Übungen

- Finden Sie heraus, ob Groß- und Kleinschreibung unterschieden wird!
- Erstellen Sie ein Muster, das auf alle phonetisch äquivalenten Schreibweisen des Eigennamens *Meier* passt!
- Erstellen Sie ein Muster, das auf die folgenden Zeichenketten passt!
  - `"***"`
  - `+.`
  - `(|||)`
- Schreiben Sie ein Programm, das alle Zeilen der Eingabe ausgibt, die *foo* enthalten!
- Schreiben Sie ein Programm, das alle Zeilen der Eingabe ausgibt, die *foo* und *bar* enthalten!

Zeichenklassen

- `[abc]` statt `(a|b|c)`
- `[0-9]`, `[a-xz]`
- `[0-9-]`
- `[\000-\037]` (Steuerzeichen)
- Klasse steht für **ein** Zeichen
- Quantoren: `[0-9]+`
- Negation: `[^aeiou]`

Zeichenklassen – Kurzformen

- `[0-9]` – `\d`
- `[\da-fA-F]`
- `[a-zA-Z0-9_]` – `\w`
- `[\r\n\t\f ]` – `\s` (Whitespace)
- `\w+\s+\w+` – *Wort* – *Zwischenraum* – *Wort*
- Negation:
  - `\d` – `\D`
  - `\s` – `\S`
  - `\w` – `\W`
  - `[\d\D]` – `.`
  - `[^\d\D]`

Anker

- Zeilenanker:
  - Zeilenanfang: `^`
    - `/^hello/`
    - aber: `/^[^helo]/`
  - Zeilenende: `$`
    - `/world$/`
    - `/^\s*$/`
- Wortanker:
  - `\W+\b\w+\b\W+`
  - `/\Bmeier\b/`: *Obermeier* aber nicht *meier*
  - `/\b\$\w+\b/` Perl-Skalarname? (oops!)

Klammern – Speicherlisten

- Klammern gruppieren
- und speichern gefundene Sequenzen
- Speicherung in einer Liste
- Ablage in der Folge der öffnenden Klammern
- Zugriff über:
  - \1, \2, ... innerhalb des Musters
  - $1, $2, ... außerhalb des Musters
- Beispielanwendung: Muster für Doppelbuchstaben.
  - /\w\w/ zu allgemein.
  - /(.)\1/

Rückwärtsreferenzen

- Referenz auf gefundene Sequenzen des Musters
- /(.)\1/
- /^(\w+)\s+\1$/
- /(hello|world)(foo|bar)\2\1/
- /((Ober|Unter)meier)\s+\2$/
- Außerhalb des Musters:
  - $1, $2, etc.
  - Listeneinträge werden bei Bemusterung überschrieben
  - möglichst nur kurz verwenden
  - sichern: if (/(\d+)/) {$n = $1}
  - Lesbarkeit

Benannte Musterfolgen

- alternativ zur Klammerposition
- /(?<Wort>\w)/
- /(?'Wort'\w)/
- /(?<Wort>\w)\k<Wort>/ – Rückwärtsreferenz
- /(?<Wort>\w)\1/ – mischbar mit Positionsreferenz
- gespeicherte Sequenzen im Hash %+
  - keys %+
  - values %+
  - for (keys %+) { print "$+{$_}"}

Reguläre Ausdrücke – Präzedenz

1. Klammern
2. Quantoren: (abc+) – (abc)+
3. Anker und Zeichenfolgen
4. Alternation

Übungen

- Worauf passt `/^foo|bar$/`
- Ein HTML-Eintrag setzt sich aus Tag und Attribut zusammen, zum Beispiel `<image source="bild.jpg">`. Statt Anführungszeichen dürfen auch Apostroph erscheinen. Welches Muster isoliert den Namen der Source in einer Speichervariablen?
- Ein anderes HTML-Konstrukt sieht den Tag am Anfang und am Ende vor, zum Beispiel `<H1>Titel</H1>`. Isolieren Sie den Eintrag zwischen Taganfang und -ende!
- Erstellen Sie ein Muster zur Erkennung gültiger Perl-Skalarnamen!
- Erstellen Sie ein Muster, das Zeilen mit doppelt auftretenden Wörtern erkennt!

Pattern Matching

- `/MUSTER/` als Kurzform für `m/MUSTER/`
- unpraktisch, wenn `MUSTER` Slashes enthält;
- alternative Muster-Begrenzer:
  - `m(MUSTER)`
  - `m!MUSTER!`
  - `m^MUSTER^`
  - `m[MUSTER]`

Bindungsoperator =~

- Matching gegen beliebige Variable
- `$line =~ /MUSTER/`
- `$line =~ m(MUSTER)`
- Fallunterscheidung: `if ( $line =~ m[MUSTER] ) ...`
- Wert des Matching-Operators: `$passt = ( $line =~ /PTN/ )`

Pattern Matching – Optionen

- Groß-/Kleinschreibung: `m/MUSTER/i`
- Sonderbedeutung von Newline aufheben: `m/MUSTER/s`
- Globales Matching: `while ( /(MUSTER)/g ) { $n += 1 }`
- Matching im Listenkontext: `@a = ( $line =~ m/(MUSTER)/g )`
- oder: `($a, $b) = ( $line =~ m/(M1).*(M2)/g )`

Interpolation in Mustern

- `$p = 'hel+o'`
- Matching: `$line =~ /$p/`
- Muster kann veränderlich sein
- Mehrfachnutzung komplizierter Muster:
  - `$num = '[+-]?\d+\.\d*([Ee][+-]\d+)?'`
  - `if ( $line =~ /$num/ ) ...`
  - `if ( $line =~ /$num/o ) ...` (Muster fix)

Substitution

- Suchen und Ersetzen
- `s/MUSTER/Ersatz/`
- Beispiele:
  - `s/(\w+)\s+(\w+)/$2 $1/`
  - `s/^/>>> /`
  - `s/hello//`
  - `s/(\w+)/\U$1/`
  - `s/\s+/_/g`
  - `s(this)[that]`

Trennen – *split*

- Trennen an Mustern
- `@f = split /sep/, $line`
- `($a, $b, $rest) = split /\s+/, $line, 2`
- `@a = split /;/, ";;;foo;bar;"`
- `@a = split '␣', $line`
- `@a = split '', $line`
- `@a = split`
- `@a = split /(\s+)/`

Verbinden – *join*

- Gegenstück zu `split`
- keine regulären Ausdrücke
- `$line = join ',␣', @a`
- `$text = join "\n", @lines`

Übungen

- Zählen Sie die Wörter einer Zeile!
- Gegeben sei ein Text (Zeilen aus Wörtern und Daten):
  - geben Sie den Text mit einem Wort/Datum je Zeile aus;
  - addieren Sie die Werte aller Zahlen;
  - erstellen Sie eine sortierte Liste mit allen großgeschriebenen Wörtern von mehr als drei Buchstaben;
  - Geben Sie die Häufigkeit jedes Wortes unabhängig von Groß-/Kleinschreibung an;
  - geben Sie alle Wörter an, die alle Vokale enthalten;
  - geben Sie alle Wörter an, die die Vokale in alphabetischer Reihenfolge enthalten;
  - geben Sie alle Wörter an, die am Ende die gleichen drei Buchstaben tragen wie am Anfang;
- In einer Datei wird das Komma als Dezimaltrenner benutzt. Ersetzen Sie nur diese Kommata durch Punkte.

Fallunterscheidung *unless* vs. *if*

```
if ( $x > 3 ) {
    # nop
} else {
    print "low␣x";
}
```

```
unless ( $x > 3 ) {
    print "low␣x";
}
```

```
unless ( $x > 3 ) {
    print "low␣x";
} else {
    print "high␣x";
}
```


Schleife *until* vs. *while*

```
while ( $n < 10 ) {
    print $n;
    $n += 1;
}
```

```
until ( $n > 10 ) {
    print $n;
    $n += 1;
}
```


Ausdrucksmodifizierer

```
if ( $n < 0 ) { print "$n␣negative"}
print "$n␣negative" if $n < 0
$n += $n until $n > 1000
&calc($_) foreach @data
print "$1␣numeric" if /(\d+)/
```

- Stilmittel (Ausnahmesituationen)
- Kontrollausdruck wird vorab ausgewertet
- Bei Codeerweiterung oft Wandlung in Standardkontrollstrukturen erforderlich

Nackte Block-Struktur

```
{
    my $x; # local scope
    $x = 3 * $y + 4;
    print "x␣=␣$x";
}
```

- *nackter* Block analog *while*-Schleife
- ein Durchlauf
- lokale Variable

Auto-In/Decrement

```
$n += 1
```

```
$n ++
```

```
$cnt{$_}++ foreach @word
```

```
$n -- if $n > 0
```

```
$x = 3;
$y = $x ++ # Postinkrement
$y = ++ $x # Präinkrement
```

```
$a[++ $i] = $i + $i ++ # Vorsicht(Sequenzpunkt!)
```


Schleife *for(;;)*

```
for ( $s = 0, $i = 0; $i < $n; $i ++ ) {
    $s += $i;
}
```

```
$s = 0;
$i = 0;
while ( $i < $n ) {
    $s += $i;
    $i ++;
}
```

```
for (;;) {} # Endlosschleife
```


Schleife *for(;;)* vs. *foreach()*

```
$n = 0;
foreach ( @data ) {
    print "$n:␣$_";
    $n ++;
}
```

```
for ( @data ) { # equivalent to foreach
    print "$n:␣$_";
    $n ++;
}
```

Schleifenkontrolle

```
while ( <> ) {
    last if /__END__/; # loop exit
    ...
} # jump here
```

```
while ( <> ) {
    s/#.*//;          # ignore comment
    next if /^\s*$/; # ignore empty line
    ...
    # jump here
}
```

```
while ( 1 ) { # jump here
    print "$x␣*␣$y␣=␣?"; $z = <>;
    redo unless $x * $y == $z;
    ...
}
```


Schleifenkontrolle mit Labels

```
LINE: while ( <> ) {
    foreach ( split ) {
        last LINE if /FINISH/; # nested loop
    }
}
```

```
if ( /desaster/ ) goto RESCUE; # remember Dijkstra
...
RESCUE: # emergency code
```


Do-Block

```
$x = do {
    $n *= 2;
    $n + 3;
};
```

- *do* evaluiert die Statements im Block und
- liefert Wert des letzten Statements zurück.

Do-Block mit Endbedingung

```
do {
    $n += 1;
    $n *= 2;
} while $n < 100;
```

- Modifier *while*
- mindestens ein Durchlauf
- *do* bildet **keine** Schleife
- kein *last*, *next*, *redo*

Logische Operatoren

```
if ( $x < 3 || $x > 7 ) {
    print "x␣von␣5␣entfernt";
}
```

```
if ( $x > 3 && $x < 7 ) {
    print "x␣nah␣bei␣5";
}
```

- `$c = $cnt{$word} || 'unseen';`
- `( $a < $b ) && ( $a = $b ) # instead of if`
- `$a < $b and $a = $b # precedence`
- Short Circuit (Teilevaluation)
- Seiteneffekte
- liefert den letzten evaluierten Ausdruck
- Lesbarkeit

Ternärer Operator

```
if ( $x % 2 == 0 ) {
    $y = $x / 2;
} else {
    $y = 3 * $x;
}
```

- `$y = ($x%2 == 0) ? ($x/2) : (3*$x);`
- Kurzform für *if-then-else*
- liefert Wert
- Lesbarkeit

Sortieren – Operatoren

```
sort { $a <=> $b } @numbers;
```

```
sort {
    length $a <=> length $b or
    $a cmp $b
} @words;
```

```
sort { # Hash sort
    $cnt{$a} <=> $cnt{$b} or
    $a cmp $b
} keys %cnt;
```


File-Handles

- File-Handle zur Identifikation einer Datei (statt Namen)
- Bezeichner für File-Handles in Großbuchstaben
- vordefinierte File-Handles: *STDIN*, *STDOUT*, *STDERR*

Dateien – Öffnen und Schließen

- `open INPUT, "data.txt"`
- `open INPUT, "<data.txt" # read`
- `open OUTPUT, ">data.txt" # write`
- `open DATA, "+<data.txt" # read AND write (binary files)`
- `open DATA, "+>data.txt" # write AND read (binary files)`
- `open LOG, ">>sys.log" # append`
- `$dat = "data.txt"; open OUT, "> $dat"`
- `close HANDLE; # asap`
- Tipp: Textdateien nicht zum Lesen *und* Schreiben öffnen

Fatale Fehler

```
my $success = open DATA, "data";
if ( $success ) {
  # process file
  close DATA;
}
```

```
open DATA, $file or
die "$0: can't open $file '$!'";
```

```
for ( @files ) {
  unless ( open DATA, $_ ) {
    warn "can't open $_: $!";
    next;
  }
}
```


Verwendung von File-Handles

```
open DATA, $file;
while ( <DATA> ) {
  ...
}
```

```
open LOG, "> $log";
print LOG "information"; # NO comma
```

```
open OUT, ">data";
printf(OUT "x = %8d\n", $x); # NO comma
```

```
open OUT, ">foo.dat";
print OUT "this";
open OUT, ">bar.dat"; # implies close
print OUT "that";
```


Standard Ausgabe-File-Handle

```
open OUT, ">data";
select OUT;
print "$x, $y, $z"; # output to OUT
```

```
select OUT;
$| = 1;
print "$x, $y"; # to OUT, implies flush
```

Standard-File-Handles

- Umleitung der Standard-I/O File-Handles erlaubt:

```
open STDOUT, ">myfile";
print "hello\n"; # goes to myfile
```

- Scheitert die Umleitung, werden Standard-I/O File-Handles automatisch neu geöffnet
- Schließen der Standard-I/O File-Handles erfolgt meist bei Service-Prozessen

Datei Tests

- `die "file␣'$file'␣exists" if -e $file`
- `open "$d" unless -d $d`
- `warn "file␣very␣old" if -M HANDLE > 100`
- `push @a, $f if -s $f > 1e6 and -A $f > 100`
- `for ( @files ) { print "$_␣lesbar" if -r }`

Datei Testoperatoren

Op.	Bedeutung
-r	Datei ist lesbar
-w	Datei ist beschreibbar
-x	Datei ist ausführbar
-o	Datei gehört dem User
-e	Datei existiert
-z	Datei ist leer
-s	Datei besitzt Inhalt (liefert Größe in Bytes)
-f	ist reguläre Datei
-d	ist Verzeichnis
-M	Dauer seit letzter Modifikation in Tagen
-A	Dauer seit letztem Zugriff in Tagen

Datei-Eigenschaften

- ```
  ($dev,$ino,$mode,$nlink,
  $uid,$gid,$rdev,
  $size,
  $atime,$mtime,$ctime,
  $blksize,$blocks)
  = stat($filename);
  ```
- ```
  $size = (stat $f)[7];
  ```
- ```
  $mtime = (stat _)[9]; # from last stat; faster
  ```
  ```

Datei-Zeitmarken

```
$mtime = (stat $file)[9];
($sec,$min,$hour,$mday,$mon,$year,
$wday,$yday,$isdst) = localtime($mtime);
```

- Monat: 0 – 11
- Jahr: ab 1900
- Wochentag: 1 – Montag
```
$t = localtime($mtime); # scalar context
```

```
$now = time; # Epoch, seconds since
             # 1970-01-01 00:00:00 UTC
```

```
$now = time;
$last_week = $now - 7 * 24 * 60 * 60;
utime $now, $last_week, $file; # atime - now
```


Binärdateien

Lesen:

```
open DAT, $file;
binmode DAT; # platform!
read DAT, $buf, $length;
close DAT;
```

Schreiben:

```
open OUT, ">$result";
binmode OUT;
print OUT $buf;
close OUT;
```

Positionieren:

```
open DAT, $file;
read DAT, $buf, $length;
seek DAT, 8, 0;  # position to byte 8
                 # 0 - from beginning
                 # 1 - relative to position
                 # 2 - from end (normally negative)
$pos = tell DAT; # actual position
```


Binärdaten wandeln

```
open DAT, $file;
read DAT, $buf, $length;
@a = unpack("C4LLNV", $buf); # Endianess
```

```
my $buf = pack("CSL", 7, 42, 12345);
```


pack/unpack – Formate

Format	Bedeutung
A	Text/Zeichenkette
c/C	signed/unsigned char
s/S	signed/unsigned short
l/L	signed/unsigned long
N	unsigned long in big Endian
V	unsigned long in little Endian
f/d	float/double in nativem Format
u	String uuencodiert
x	Null Byte

Übungen

- Schreiben Sie ein Programm, dass die Dateigrößen aller Files aus *ARGV* bestimmt, addiert und ausgibt.
- Sortieren Sie die Dateien aus *ARGV* nach Alter. Bei gleichem Alter sortieren Sie alphabetisch.
- Erstellen Sie eine Binärdatei, in der Sie den Wert der Zahl 42 im nativen Format speichern. Lesen Sie diese Datei auf einer Plattform mit anderer Endianess ein. Welchen Wert enthält die Datei?
- PNG-Dateien enthalten ab der Byteposition 16 die Breite und die Höhe des Bildes als big endian long. Schreiben Sie ein Programm, das für gegebene PNG-Dateien die Bildbreite und -höhe ausgibt. Überprüfen Sie, ob in den ersten 16 Bytes der String *PNG* steht!

Dateisystem – Navigation

```
chdir "/bin" or
die "can't␣cd:␣$!";
```

```
use Cwd; # package
my $dir = getcwd; # where am I?
```

```
use Cwd 'realpath';
my $abs_path = realpath $dir;
```

- aktuelles Verzeichnis festlegen/abfragen
- Schreiben Sie ein Programm `cd.pl`. Kann es das Kommando `cd` ersetzen?

Verzeichniseinträge – Globbing

- `@files = glob("*.txt")`
- `@files = glob("/[a-m]?[f-z]*/??[a-g]*.*")`
- relativer/absoluter Pfad
- Unterschied zu regulären Ausdrücken
- (alte)alternative Syntax: `@files = <*.txt>`

Directory-Handles

```
opendir D, "/home" or die "can't␣open␣dir:␣$!";
for ( readdir D ) {
  print "$_";
}
rewinddir D; # once again
...
closedir D;
```

- effizienter als Globbing
- keine Sortierung

Verzeichnis-Traversierung

```
use File::Find;
sub wanted {
  print if -s $_ > 1000;
}
find \&wanted, "/home";
```

- rekursive Traversierung
- Verwaltung vieler offener Directory-Handles

Dateien löschen

- `unlink "data.txt";`
- `unlink @files;`
- `unlink "directory" or warn "use␣rmdir";`

Dateien umbenennen

-
```
for $old ( @files ) {
  my $new = $old;
  $new =~ s/ //g; # delete white space
                  # from file name
  next if $new eq $old;
  rename $old, $new;
}
```

-
```
rename "file", "/far/away"; # move
```


Verzeichnisse erstellen, löschen und modifizieren

- `mkdir "dir", 0755; # Permissions`
- `rmdir @dirs; # if empty`
- `chmod 0664, "file.dat";`
- `chown $user, $group, @files`

Verzeichniseinträge – Namensbestandteile

- ```
  use File::Basename;
  $fullname = basename $file;
  $name = basename $file, ".txt", ".dat";
  $path = dirname $file;
  ```
- ```
  use File::Basename qw/ /; # don't import anything
  $fullname = File::Basename::basename $file;
  ```


Namen plattformunabhängig – OO-Style

```
use File::Spec;
my $file = "work.txt";
my @dir  = qw/ home user data /;
my $fullname = File::Spec->catfile(@dir, $file);
```

- Methodenaufruf: *Package* -> *Methode*
- Dokumentation: `perldoc File::Spec`

Übungen

- Schreiben Sie ein Programm, dass die Dateien löscht, welche über die Kommandozeile angegeben sind.
- Schreiben Sie ein Programm, das
  - eine Datei umbenennt, wenn zwei Dateinamen gegeben sind;
  - Dateien in ein Zielverzeichnis verschiebt.
- Schreiben Sie ein Programm, das
  - aus den gegebenen Dateinamen Leerzeichen entfernt,
  - Groß- in Kleinbuchstaben wandelt.
- Schreiben Sie ein Programm, dass für alle gegebenen Dateien ein bestehendes Suffix gegen einen neuen Extender austauscht.

Programm-Aufrufe *system*

- Aufruf externer Programme
- aufgerufene Programme nutzen Standardd-I/O-Kanäle des rufenden Prozesses
- `system("date");`
- `system("command␣@args"); # per Shell`
- `system("zip␣archiv␣*.txt"); # globbing by Shell`
- `system("command", @args); # direct call`
- `system("zip", "archiv", "*.txt"); # oops`

Kommandoausgaben – Backquotes

- Weiterverarbeiten der Ausgabe externer Programme
- `$now = `date`;`
- `@lines = `perldoc -t -f sin` # Liste;`
- `system()` verwenden, wenn Ausgabe nicht interessiert
- Problem bei Dialogprogrammen

Environment

- Umgebungsvariable des Prozesses: *PATH* usw.
- Zugriff per Hash *ENV*:
  - `$path = $ENV{'PATH'};`
  - `delete $ENV{'PATH'};`
  - `$ENV{'PATH'} .= ":."; # dangerous`
  - `$ENV{'NOTE'} = 'interessant';`
- Vererbung an gerufene Prozesse

Prozesse als File-Handles

- Standard-I/O von Prozessen als File-Handles;
- Von Fremdprozess lesen:

```
open D, 'date|'; # launch process
$now = <D>;
close D;          # terminate other process
```

- Zum Fremdprozess schreiben:

```
open M, '|mail';:
print M "subject: Perl\n";.
```

-
```
open P, '|prog|'; # Oops - only ONE pipe!
```
- Unterschied zu Backquotes:

```
open F, "find / -type f|"; # run as separate process
while ( <F> ) {
  print "found file $_\n";
}
```


Netzwerk – TCP-Client

```
#!/usr/bin/perl -w
use IO::Socket;

$remote = IO::Socket::INET->new(
    Proto    => "tcp",
    PeerAddr => shift || "localhost",
    PeerPort => shift || 2345,
    )
    or die "cannot connect";
while ( <$remote> ) { print }
```

Netzwerk – TCP-Server (multithreaded)

```
#!/usr/bin/perl -w
use IO::Socket;
$server = IO::Socket::INET->
    new( Proto     => 'tcp',
         LocalPort => shift || 2345,
         Listen    => SOMAXCONN,
         Reuse     => 1 ) or die "can't setup server";
while ($client = $server->accept()) {
    if ( $pid = fork ) { # parent or child?
        close $client; next;
    }
    printf "Connect from %s\n", $client->peerhost;
    while ( 1 ) {
        print $client localtime() . "\n"; sleep 1;
    }
}
```


Übungen

- Rufen Sie das Programm *date* aus einem Perlprogramm heraus aus!
- Lösen Sie die gleiche Aufgabe, nachdem Sie die *PATH*-Variable in Ihrem Perl-Programm gelöscht haben!
- Lassen Sie Ihr Programm die Ausgabe des Kommandos *date* interpretieren und den aktuellen Tag des Monats ausgeben!
- Erweitern Sie den TCP-Server so, dass er vom Client die Verzögerungszeit in der Schleife entgegennimmt. Diese Zeit sollte nie Null werden! Warum?

Eval

- Block Evaluation – Fehler abfangen
  - fatale Fehler:
    `eval { $x / $y }; warn $@ if $@`
  - Syntaxfehler zur Laufzeit:
    `$re = '[abc)'; eval { m/$re/ };`
- String Evaluation
  - ausnutzen der zentralen Interpreterfunktion
  - de facto selbstmodifiziereder Code – problematisch
  - erzeugen von Variablen zur Laufzeit:
    `$cmd = '$x = 42'; eval $cmd;`
- symbolische Referenz:
  `$r = 'foo'; $$r = 42; print $foo;`

Übungen

- Erzeugen Sie mittels *eval* die Variablen $a – $z und belegen Sie diese mit 1–26!
- Lösen Sie die vorherige Aufgabe mittels symbolischer Referenzen!

GUI – Hello World

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

use Tk;   # borrowed from TCL/Tk
          # implemented by
          # Nick Ing Simmons

$top = MainWindow->new; # OO-style
$hello = $top->Button(
    -text => "hello",
    -command => sub{ print "Ciao\n"; exit },
); # attributes in a hash
$hello->pack; # geometry management

MainLoop;     # event handling
```

Editor – Methode *Datei laden*

```perl
$file = "empty";
sub fsel {
  $file = $top->FileSelect->Show;
}
sub load {
  my ($t) = @_;
  $t->delete( "1.0", "end" );
  fsel;
  if ( !open (FH, $file)) {
    my $d = $top->Dialog( -text => "Tja!" )->Show;
  } else {
    while (<FH>) { $t->insert("end", $_); }
    close FH;
  }
}
```


Editor – Methode *Datei sicheren*

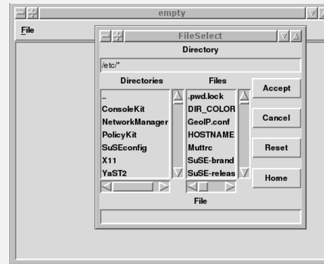
```perl
sub save {
  my ($t) = @_;
  open (FH, ">$file");
  print FH $t->get("1.0", "end");
  close (FH);
}

sub saveas {
  my ($t) = @_;
  fsel;
  save $t;
}
```


Editor in 99 Zeilen

```perl
use Tk;
use Tk::FileSelect;
$top = MainWindow->new;
$top->title($file);
$menu = $top->Frame( -relief => 'raised',
                     -borderwidth => 2 );
$menu->pack(-fill => 'x');
my $f = $menu->Menubutton( -text => 'File',
                           -underline => 0 );
$f->command( -label => 'Open',
             -command => sub { load $text } );
$f->command( -label => 'Save',
             -command => sub { save $text } );
$f->command( -label => 'Save␣as',
             -command => sub { saveas $text } );
# ...
```

Editor – Hauptschleife




```
# ...
$f->separator;
$f->command( -label => 'Quit',
    -command => sub { exit 0 } );
$f->pack(-side => 'left');
$text = $top->Scrolled( "Text",
    -scrollbars => 'oe',
    -width => 80, -height => 25,
    -font => '-*-courier-*--18-*' )->pack;
MainLoop;
```


Übungen

- Erstellen Sie ein Programm, das Radiobuttons anbietet, um seine Hintergrundfarbe einzustellen!
- Erstellen Sie ein Programm, das die aktuelle Uhrzeit darstellt! Die Dauer zwischen den Aktualisierungen soll über einen Slider einstellbar sein.

Betriebssystem-Kommandointerpreter

- Bourne-Shell `sh` – Posix-Standard
- Eingabeaufforderung `command.exe`
- `bash`

Universelle Script-Sprachen

- Python
- Ruby
- Lua
- Lisp/Scheme
- Tcl
- PHP

Spezialisierte Script-Sprachen

- Gnuplot – Diagramme, Funktionsverläufe
- Matlab/Octave – Numerik
- Maple/Maxima – Computer-Algebra-Systeme
- AWK, sed – Utilities
- AGC/DSKY – Bordrechner im Apollo-Programm

Makro Script-Sprachen

- VBA
- Makrosteuerung in CAD-Systemen

Übungen

- Schreiben Sie *Hello World*-Programme in Scriptsprachen Ihrer Wahl!
- Vergleichen Sie verschiedene Scriptsprachen hinsichtlich:
  - Ausführungsgeschwindigkeit;
  - Funktionsumfang;
  - Typisierung;
  - Dokumentation;
  - Größe des Interpreters!