Perl Grund-lagen

© Bernd Klein

Der Kopf hinter Perl:
Larry Wall

Er konzipierte und schrieb Perl.


Larry
Wall

History of Perl

Perl 1.0
18. Dezember 1987

Perl 2.0 (1988)
Bessere reguläre Ausdrücke

Perl 3.0 (1989)
Unterstützung für binäre Datenströme

Programming
Perl
de-facto-Referenz
1991

Perl 4.0 (1993)
sehr kurzlebig

Perl 5.0 (1994)
komplett neuer Interpreter

Perl 5.10.0
(18. Dezember 2007)

Perl ist ein Akronym

Practical Extraction and Report Language

Perl nie in Großbuchstaben schreiben.

Family

Perl ist eine dynamsiche Programmiersprache, die von vielen anderen Sprachen geborgt hat, wie z.B.

C
Bourne and other Shell scripting languages
AWK
Sed
Lisp

Die sieben Besonderheiten von Lisp

- Integrierte Unterstützung für Listen
- Automatische Speicherverwaltung
- Dynamische Typisierung
- First-Class Funktionen
- Einheitliche Syntax
- Interaktive Umgebung
- Erweiterbarkeit
- History

Merkmale

Perl ist im Prinzip eine prozedurale Sprache ähnlich wie C mit
- Variablen
- Ausdrücken
- Zuweisungen
- Code-Blocks in geschweiften Klammern
- Kontrollstrukturen
- Unterprogramme

Besondere Strukturen um das Arbeiten beim Parsen, bei der Textverarbeitung und bei anderen Data-Management -Aufgaben zu erleichtern:
- **Listen, ähnlich wie in Lisp**,
- **Assoziative Arrays** (Hashes) wie in AWK,
- **Reguläre Ausdrücke** wie beim sed.

Merkmale, Fortsetzung

Mit Perl 5 kamen zusätzliche Komponenten zu Perl:

- Komplexe Datenstrukturen,
- first-class Funktionen (d.h. closures als Werte),
- Objekt-orientiertes Programmier-Modell. (Referenzen, Packages, Klassen-basierte Methoden und anderes)
- Compiler-Direktiven (z.B. das “strict pragma”).
- Die Möglichkeit Code in wiederverwendbare Module zu packen.

Implementierung

Perl ist eine interpretierte Sprache

The Kern-Interpreter ist in C geschrieben.

Zusätzlich gibt es eine große Sammlung von Modulen, die sowohl in C als auch in Perl geschrieben sind.

12 MB als gepacktes tar file.

- kompiliert über1 MB
- 150,000 Zeilen C-Code.
- ca. 500 Module existieren in der Perl-Distribution, was etwa 200.000 Zeilen Perl und etwa 350,000 Zeilen Code entspricht.

Perl starten

- In einer Kommandozeile mit:

```
perl -e 'print "Hello, world\n"'     #Unix
perl -e "print \"Hello, world\n\""   #Windows
```

- Wie eben aber die Eingabe über den Standard-Input-Stream:

```
echo "print 'Hello, world'" | perl -

oder:
% perl
print "Hello, world\n";
^D
```

Übung

Testen Sie die folgende Kommando-Zeile in einer Shell:

```
perl -e 'print "Hello, world\n"'
 oder
echo "print 'Hello, world'" | perl -
```

“Hello World” in Java

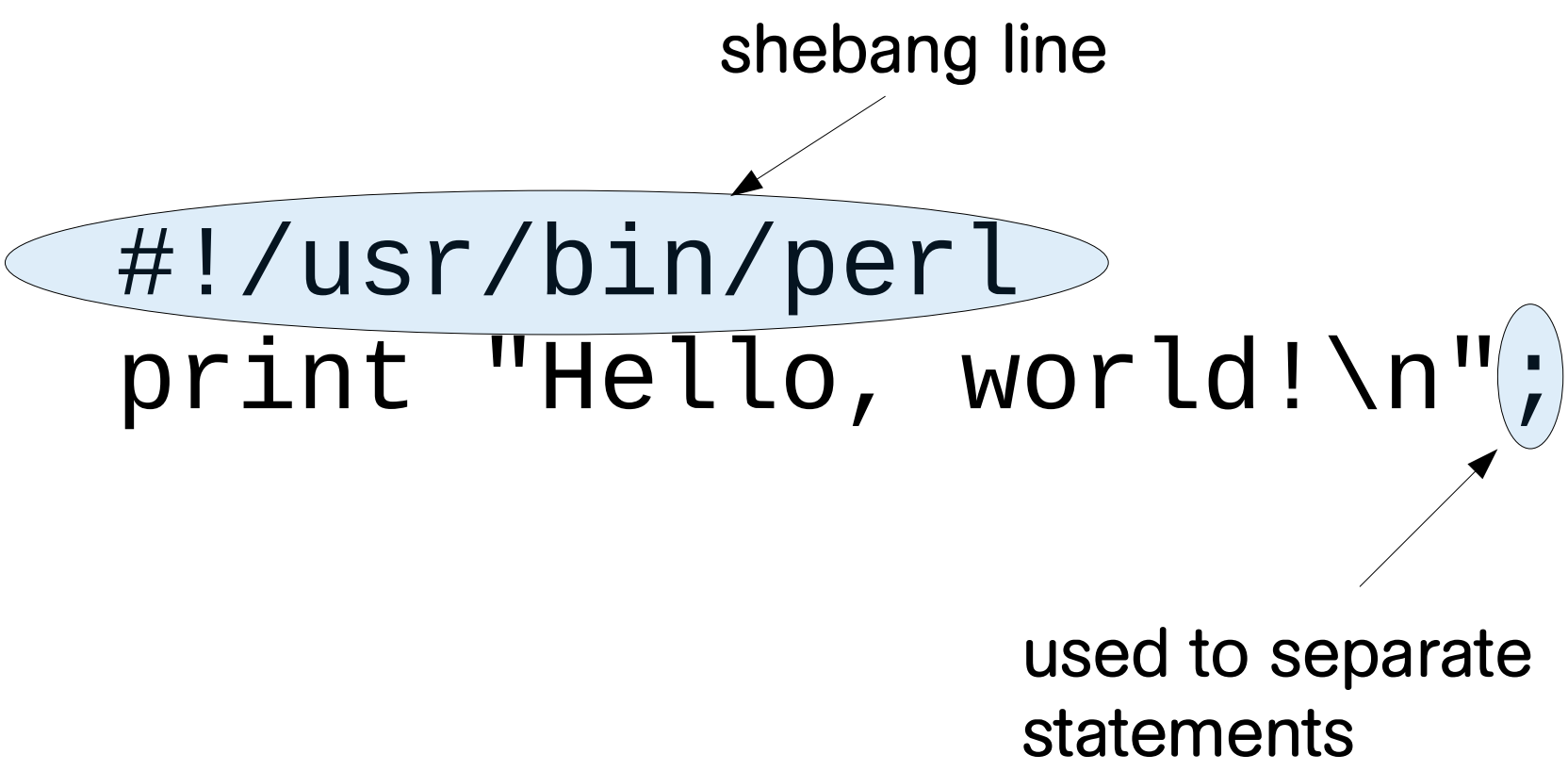
Der beschwerliche Weg “Hello World”:

```
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //
Display the string.
    }
}
```

Perl-Dateien starten

- lperl testpgm
- Auf einem Unix-System: #!-Zeile.

Hello, world!


shebang line
#!/usr/bin/perl
print "Hello, world!\n";
used to separate statements

Übung

Schreiben Sie in einem Editor das „Hello Word“-Programm und speichern Sie es als hello.pl ab.

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

Aufruf des Programms mitl:

```
perl hello.pl
```

Mit folgender Zeile wird das Programm ausführbar gemacht:

```
chmod a+x hello.pl
hello.pl
```

Wie stellt man Fragen ...

... und wie erinnert man sich an die Antworten/ Ergebnisse!

Input und Output

Wir schreiben nun eine personalisierte Version von „Hello World“, die den Benutzer und nicht die ganze Welt grüßt:

Dazu benötigen wir eine Variable, die den Namen aufnehmen kann:
Wir benutzen $name, ein Skalar (einzelner Wert).

Mit dem <STDIN>-Konstrukt können wir eine Zeile von der Konsole lesen. (<> Diamant-Operator)

```
print "Wie heißen Sie? ";
$name = <STDIN>;
```

Übung

Speichern Sie das vorige Skript unter
`input1.pl`

```
print "Wie heißen Sie? ";
$name = <STDIN>;
```

Starten Sie das Programm!

Fortführung des Beispiels

Das Programm soll den Namen zusammen mit „Hallo“ ausgeben:

```
print "Hallo $name!\n";
```

Hängt man obige Zeilen an die vorigen Zeilen an, macht das Programm folgendes:

```
Wie heißen Sie? Larry
Hallo Larry
!
```

Kleiner Schönheitsfehler

Das Ausrufezeichen erscheint erst in der nächsten Zeile und nicht direkt hinter dem Namen.

Der Grund:
An `$name` hängt noch ein Newline „\n“.

`chomp` ist eine Funktion, die einen Skalar als Arbument nimmt und ein eventuell vorhandenes Newline am Schluss entfernt.

Das komplette Skript

```
#!/usr/bin/perl -w
print "Wie heißen Sie? ";
$name = <STDIN>;
chomp ($name);
print "Hallo $name!\n";
```

Kommandozeilen-Optionen

Die Kommandozeilen-Optionen kommen als erstes (nach perl) auf der Kommandozeile.

Der nächste Begriff ist überlicherweise der Name des Skriptes (Dateiname) gefolt von zusätzlichen Argumenten.

Some Perl Switches

--	erzwungenes Ende der Optionen
-c	Überprüfung der Syntax des Skriptes ohne dieses auszuführen.
-d	starten des Skriptes mit dem Debugger
-h	Übersicht der Kommandozeilen-Optionen
-v	Versionsnummer von Perl
-w	Warnungen über nicht nicht zugewiesene Variablen, mehrfach definierte Funktionen usw..

Perl in C mit `perlcc` übersetzen

```
perlcc HelloWorld.pl -o Hello
```

Dieses Kommando erzeugt ein ausführbares Programm

Man kann aber auch nur C-Code mit der Option -S erzeugen:

```
perlcc -S HelloWorld.pl
```

Übung

Erzeugen Sie den C-Code für das vorige “Hello World”-Beispiel.

Variablen

Eine Variable beginnt immer mit einem speziellen Zeichen, dass ihren Typ bezeichnet:

```
$, @, or %
```

Nach einem solchen Zeichen folgt ein Unterstrich oder ein Buchstabe gefolgt von einer beliebigen Folge von Buchstaben, Ziffern und Unterstrichen. Allerdings darf/sollte die Länge nicht 255 Zeichen übersteigen.

Achtung: Perl unterscheidet zwischen Groß- und Kleinbuchstaben!

Undefinierter Wert: undef

Bevor einer Variablen ein Wert zugewiesen worden ist, hat sie den undef-Wert

Werden solche Variablen als Skalare benutzt, werden sie von Perl in einen leeren String oder den Wert Null gewandelt, je nach Umgebung.

Zuweisungen

Der Zuweisungs-Operator (=) weist einer Variablen auf der linken Seite den ausgewerteten Wert der rechten Seite zu:

```
$count = 0;
$res = $b * ($a + 2.4534);
```

Wert einer Zuweisung

Eine Zuweisung hat auch einen Wert, nämlich den Wert der rechten Seite, also der zugewiesene Wert.

Beispiel:

```
$a = 1 + ($b = 7);
```

Der Wert 7 wird der Variablen $b zugewiesen, dann wird 7 zu 1 addiert und das Ergebnis 8 wird der Variablen $a zugewiesen.

Datentypen

- ein **Skalar:** ein Zahl, ein String oder eine Referenz
- Ein **Array** ist eine geordnete Sammlung von Skalaren.
- Ein **Hash**, oder ein assoziatives Array ist eine Abbildung von Strings in Skalare; die Strings werden keys und die Scalare Werte genannt.
- Ein **file handle** ist eine Abbildung auf eine Datei, ein Gerät oder eine Pipe um zu schreiben zu lesen
- Eine **Subroutine** ist Stück Code, dass ausgeführt werden kann, dem Parameter übergeben werden können, und die einen Wert zurückgeibt.

Zeichen zur Identifikation von Variablen

```
$foo  # ein Skalar

@foo  # ein Array

%foo  # ein Hash

FOO   # ein file handle oder eine
        Konstante

&foo  # Unterprogramm (subroutine);
        das & ist optional
```

Skalare Daten

A skalar ist eine einfache Variable d.h. die einfachste Art in Perl.

Ein Skalar kann sein:

- eine Zahl (z.B. 7 oder 8.2620) oder
- ein String (wie „`hello`“).

Skalare Variablen

Eine Variable ist ein Behälter, der einen Wert enthält.

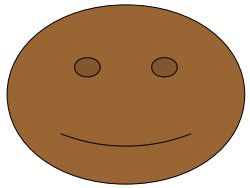
Der Name einer Variablen ist im Programm konstant, aber ihr wert kann sich während des Programmablaufes immer wieder ändern.

Eine Skalarvariable enthält eine skalaren Wert, also eine Zahl, ein String oder eine Referenz.

Sie beginnen mit einem Dollarzeichen.

Übung

Was ist der Wert von $a nach der Zuweisung?



```
$a = 024 + 026;
```

Zahlen

```
42
-42
3.1415
4.35E32
0xff          # hexadezimal
012           # oktal
23_234_767    # Unterstrich zur besseren
                Lesbarkeit
```

Internes Zahlenformat

Intern arbeitet Perl nur mit
double-precision floating-point Werten.

Es gibt also keine Integer in Perl!

Deshalb gibt es auch keine speziellen Integer-Operationen.

Fließkomma Literale

Ein Literal ist die Darstellung eines Wertes im Source Code des Perl-Programmes, also eine Konstante.

Float Literals

```
1.75
7.25e45
-6.5e24
-12e-24
-1.2E-23
```

Integer Literals

```
1
2
-1
42
1_453_987_112
```

Strings

Ein String besteht aus einer Folge von Zeichen.

Jedes Zeichen ist ein 8-Bit-Wert, damit hat man 256 verschiedene Zeichen.
(im Gegensatz zu C gibt es keine besondere Bedeutung für das NUL-Zeichen.)

Der kürzest mögliche String ist der leere String, er enthält keine Zeichen.

Die Länge eines Strings wird von Perl nicht begrenzt. Lediglich durch die Größe des Speichers.

Strings in doppelten Anführungszeichen

In einem String in doppelten Anführungszeichen, kann der Backslash dazu benutzt werden besondere Kontrollzeichen, wie z.B. newline \n, zu spezifizieren.

Beispiele von Strings in “”

```
"hello world\n"

"umlaut \334"     # umlaut ü (oktal)

"km\t1089"        # ein tab in einem
                    String
```

Strings in einfachen Anführungszeichen

Anführungszeichen markieren den Anfang und das Ende eines Strings, aber sie sind nicht Teil des Strings selbst!

Innerhalb von einfachen Anführungszeichen sind alle Zeichen erlaubt. Backslashes haben keine besondere Bedeutung.

Beispiele von Strings in einfachen Anführungszeichen

```
'Perl'       # vier Zeichen: P, e, r, l

'don\'t'     # Fünf Zeichen:
               d, o, n, single-quote, t

''           # the empty/null string

'home\\eve   # includes one backslash

'hello\n'    # not a newline!
```

String Konvertierungen

Perl wandelt automatisch wenn notwendig zwischen Strings und Zahlen um

Was druckt das folgende Skript:

```
$fruit = "3 peaches";
$vegetable = "2 cabbages";
print $fruit + $vegetable;
```

Es wird ‘5‘ ausgegeben.

Übungen

- Schreiben Sie ein Programm, dass zwei Zahlen in separaten Eingabezeilen einliest und das Produkt der beiden Zahlen ausdruckt.

Operatoren für Skalare

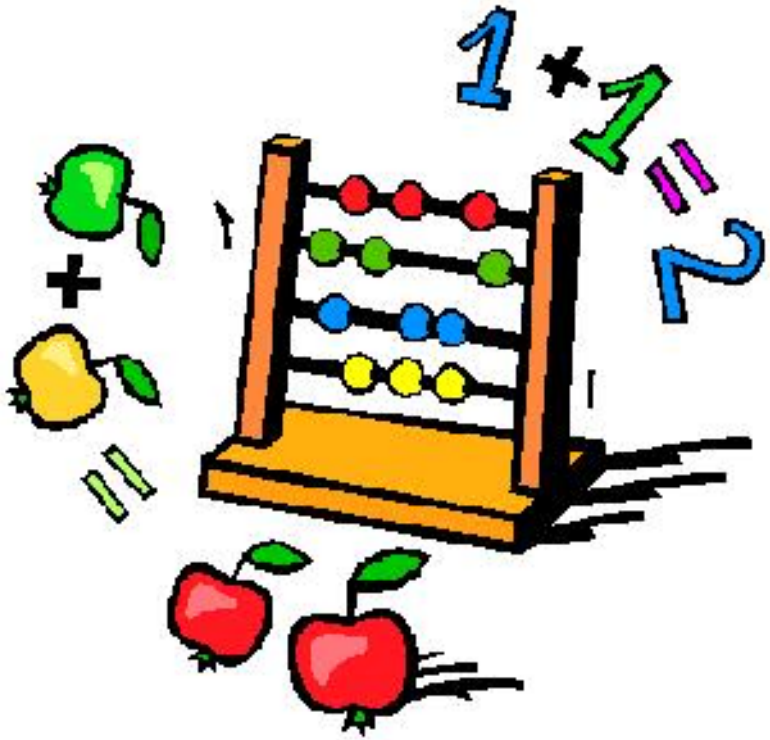
Ein Operator erwartet entweder numerische oder String-Operanden oder eine Kombination aus beiden.

Steht ein String, wo ein numerischer Wert erwartet wird, dann wird dieser automatisch in einen numerischen Wert umgewandelt. Ebenso wird ein numerischer Wert bei Bedarf in einen String gewandelt.

Numerische Operatoren

Die üblichen Operatoren für

- Sddition: +
- Subtraktion: -
- Multiplikation *
- Division /



Andere Operatoren:

Exponentiation: * *

Modulus: %

logische Vergleichsoperatoren: <, <=, ==, >=, >, !=

Präzision

Was ergibt 7.1 – 1.3 ?

5.8

Sind Sie sicher?

Ist es nicht vielleicht ...

5.79999999999998223643161

perl -e ‘printf(”%.25f\n”, 7.1 - 1.3)‘;

Operatoren für Strings

KonKatenation: “.“ Operator

Wenn man Strings mit dem “.” Operator konkateniert ändert das nicht die Werte die involvierten Strings.

Beispiele:

```
“homer” . “ “ . “simpson”

“hello world” . “\n”
```

String-Vergleiche

Equal	`eq`
Not equal	`ne`
Less than	`lt`
Greater than	`gt`
Less than or equal to	`le`
Greater than or equal to	`ge`

String Wiederholungs-Operator

Der Buchstabe x bezeichnet den String Wiederholungs-Operator. Der String auf der linken Seite des “x” wird so etwas konkateniert, wie es der numerische Wert auf der rechten Seite bestimmt.

```
“Homer” x 3          → “HomerHomerHomer”
“Marge” x (1 + 1)    → “MargeMarge”
(3 + 2) x (2.3 *3)   → “555555”
```

Übung

Was ist der Wert des folgenden Vergleichs?

```
8 < 30
```

```
true
```

```
8 lt 30
```

```
false
```

Listen und Arrays

Eine Liste stellt geordnete skalare Daten dar. Ein Array ist eine Variable, die solch eine Liste enthält.

Jedes Element eines Arrays ist eine separate Skalarvariable mit einem eigenen skalaren Wert.

Die Elemente sind durch Indices angeordnet: 0,1,2, ...

Arrays können beliebig viele Elemente haben.

Das kleinst mögliche Array hat keine Elemente, das größtmögliche wird durch die Größe des Speichers bestimmt.

Arrays

Die Werte einer Liste können sein:

- Zahlen
- Strings, oder sogar
- ein anderes Array.

Array-Variablen-Namen beginnen immer mit @

Beispiele von Arrays

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

@A = (" or ", " not ");
@prefix = (2,"B ");
@suffix = @prefix;
@william = (@prefix, @A);
@william = (@william, @suffix);

print “@william\n”;
```

List-Konstruktur-Operator

Ein Listenkonstruktor-Operator wird durch zwei skalare Werte gebildet, die durch zwei nebeneinander liegende Punkte getrennt sind.

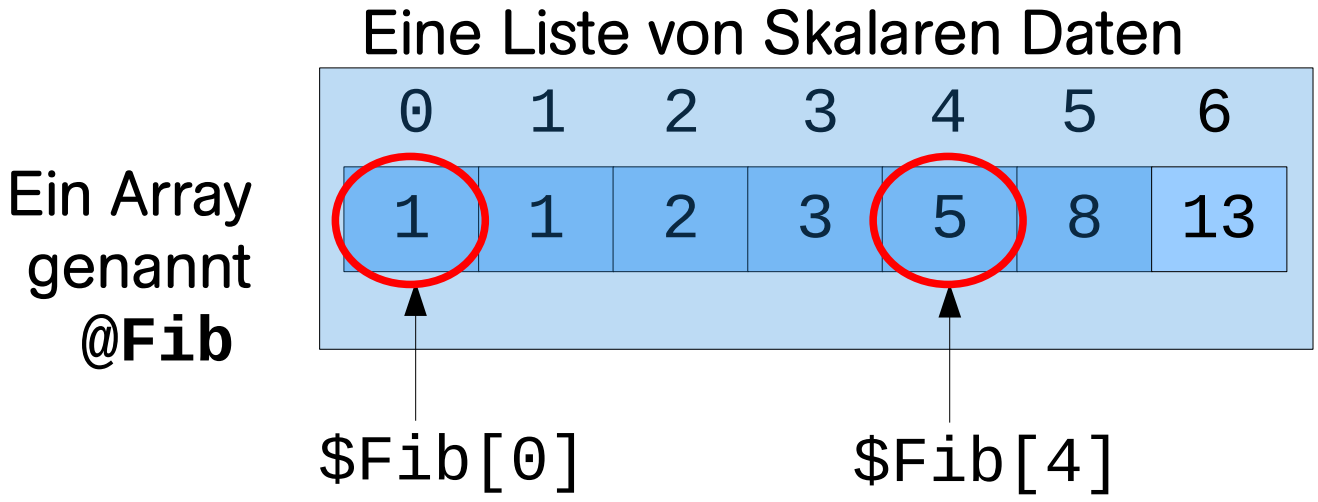
Eine Liste von Werten wird dadurch gebildet, dass mit dem linken Skalar begonnen wird und der nächste Wert jeweils durch Addition von eins gebildet wird und zwar solange bis der rechte Wert überschritten wird.

```
(1 .. 5)        ->  (1,2,3,4,5)

(2.3 .. 5.3)    ->  (2.3,3.3,4.3,5.3)

(2.3 .. 6.2)    ->  (2.3,3.3,4.3,5.3)
```

Zugriff auf ein Array



Das @ des Arraynamens wird zu einem $ bei der Referenz eines Elementes!

Übung

Was wird im folgenden Skript ausgegeben?

```
@Fib = (1, 1, 2, 3, 5, 8, 13, 21)

$i = 5;
$a = $Fib[4];
$b = $Fib[$i]++;
$c = $Fib[$i++];
$i = 5;
$d = $Fib[++$i];
print "$a, $b, $c, $d\n";
print "@Fib\n";
```

Syntaktischer Zucker für Listen-Literale

Statt Arrays mit folgenden Listen zu definieren:

```
@languages = (“english”, “french”,”german”);
```

kann man auch die “quote word” Funktion benutzen:

```
@languages = qw(english french german);
```

Slice

Ein Slice ist gewissermaßen eine Unterliste:

```
@Fib = (1, 1, 2, 3, 5, 8, 13, 21);
($Fib[1],$Fib[2]) = ($Fib[2],$Fib[1]);
print "@Fib\n";
```

Syntaktischer Zucker:

```
@Fib[0,1] = @Fib[1,0] # swap the first two el.
@Fib[0,1];            # same as
          #           ($Fib[0],Fib[1])
@Fib[0,1,2]=@Fib[1,1,1]# first three like 2nd
```

Grenzenlose Arrays

Weist man einen Wert einem Array-Element mit einem Indes jenseots der bereits existierenden unter Auslassung anderer Indizes, so werden die ausgelassenen Elemente mit dem erst “undef” initialisiert.

```
@b = (“to”, “be”);
$b[2] = “or”;
$b[5] = “be”;

print “@b\n”;
```

`$b[3]` and `$b[4]` will have the value “undef”.

Arrays können als Stapel angesehen werden

push und pop

Ein Array kann wie ein Stapel von Informationen benutzt werden.
Neue Elemente werden von der rechten Seite entnommen und dort hinzugefügt.
Dazu gibt es die Funktionen pop und push:

```
push(@list, some_value);
```

ist äquivalent zu

```
@list = (@list, some_value);
```

pop entnimmt das letzte Element einer List (Array):

```
$last_element = pop(@list);
```

Shift und Unshift

The push and pop functions operate from the ”right” side of a list, i.e. the one with the highest subscripts.

Die Funktionen shift und unshift sind analog zu pop (shift) und push (unshift).

Shift und Unshift, Beispiel

```
@colors = ("red", "green","blue");
$y = "yellow";
unshift(@colors, $y, "white", "black");
print "colors: @colors\n";
$most_left = shift(@colors);
print "most_left: $most_left\n";
print "colors: @colors\n";

# entferne den am weitesten links stehenden
# Ausdruck füge ihn auf der rechten Seite
# wieder an:
push(@colors, shift(@colors));
print "colors: @colors\n";
```

Die reverse-Funktion

Die reverse-Funktion liefert eine Liste in umgekehrter Reihenfolge zurück.


@simpsons = ("Lisa","Homer","Bart","Maggie","Marge");

```
@simpsons = ("Lisa","Homer","Bart","Maggie","Marge");
@simpsons  = reverse( @simpsons );
```

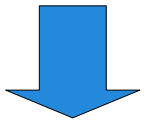
sort Funktion

Die sort-Funktion sortiert ihre Argumente in aufsteigender ASCII-Reihenfolge.

sort liefert die sortierte Liste als Ergebnis ohne die ursprüngliche Liste zu verändern.

```
@simpsons = sort("Homer","Marge","Bart","Lisa","Maggie");
print "in alphabetical order: @simpsons\n";
```

```
@Fib = (1, 1, 2, 3, 5, 8, 13, 21);
@sorted = sort(@Fib);
print "@sorted\n";
```



```
1 1 13 2 21 3 5 8
```

Die chomp-Funktion

Die `chomp`-Funktion arbeitet sowohl auf Variablen-Namen als auch auf skalaren Variablen.

Falls es sich bei dem letzten Zeichen des Strings um ein RETURN handelt, wird es entfernt.

Alle anderen Zeichen bleiben von chomp unberührt.

```
@simpsons=("Lisa\n","Homer\n","Bart","Maggie\n","Marge");
@simpsons  = chomp(@simpsons );
```

`@simpsons` sieht nun wie folgt aus:
`("Lisa","Homer","Bart","Maggie","Marge")`

The chop Function

Die chop Funktion ist ähnlich wie chomp, aber sie hackt immer “chops off” das letzte Zeichen ab.

<STDIN> liefert Array

<STDIN> liefert in einem Skalar-Kontext die nächste Zeile zurück.

In einem Listenkontext liefert <STDIN> jedoch alle verbleibenden Zeilen bis zum Dateieinde als Liste zurück. Jede Zeile entspricht einem Element der Liste.

```
@a = <STDIN>;
print "@a\n";
```

Übung

Schreibe ein Skript, dass eine Liste von Strings einliest und in separaten Zeilen in umgekehrter Reihenfolge wieder ausgibt.

Hashes

Hashes sind meist besser als **assoziative Arrays bekannt.**

Ein Hash wird wie ein Array definiert, aber man benötigt jeweils zwei Array-Elemente um ein Hash-Element zu definieren.

Werte eines Hashes werden mit geschweiften Klammern {} um den Index-Key angesprochen..

Beispiel:

```
%birthdays = ("Mike", "Apr 29", "Jane", "Feb 1",
"Freddy", "Dec 7");
print "Freddys birthday is:" . $birthdays{"Freddy"}
. " \n";
```

Übung

Schreiben Sie ein Programm, dass den Benutzer nach seinem Vornamen fragt und dann den zugehörigen Nachnamen ausgibt.

Kontrollstrukturen

- Anweisungsblocks
- Bedingte Ausführung: if/unless-Anweisung
- Schleifen:
  - while/until Anweisung
  - for Anweisung
  - foreach Anweisung

Anweisungsblocks

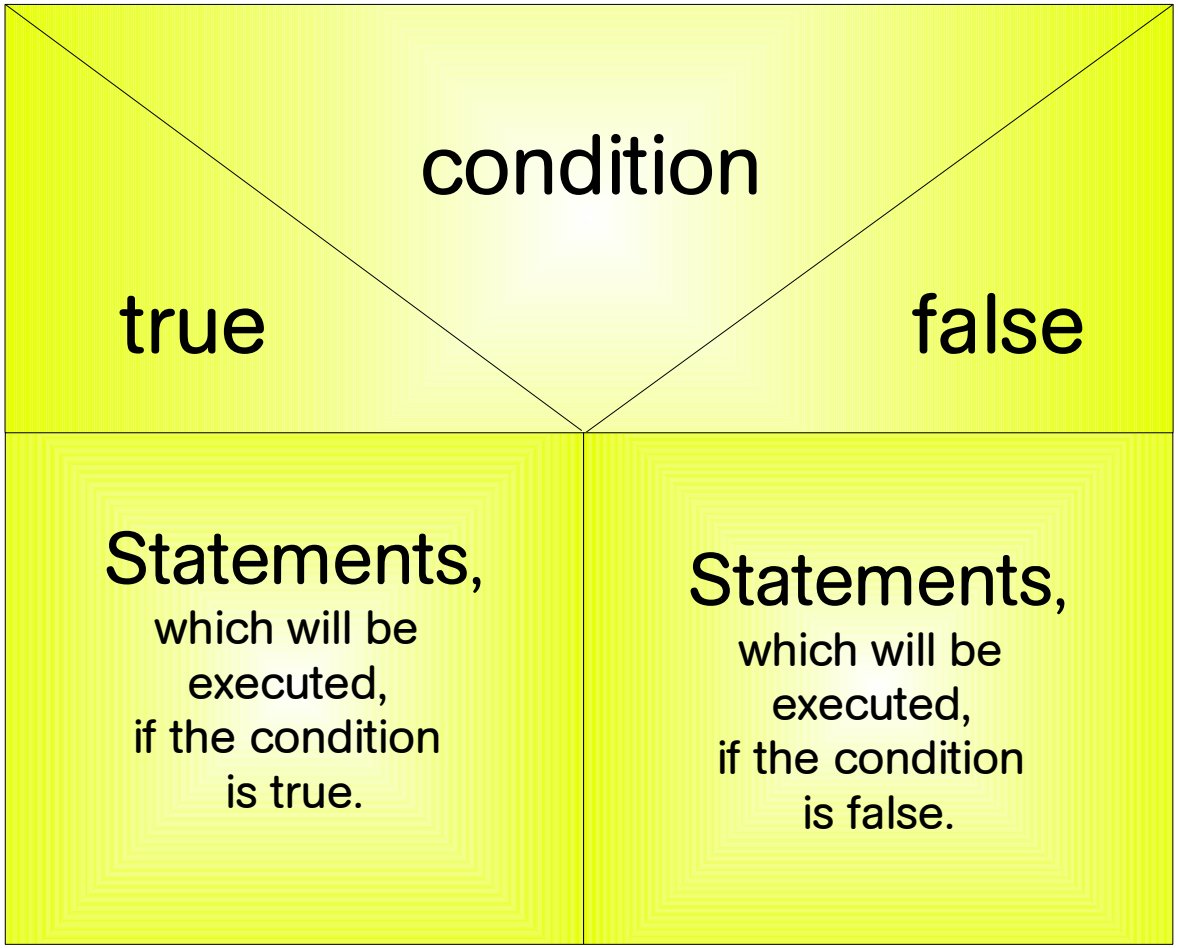
Ein Anweisungsblock ist eine Folge von Anweisungen, die von geschweiften Klammern umgeben ist.

```
{
    first_statement;
    second_statement;
    third_statement;
    ...
    last_statement;
}
```

Die Anweisungen des Anweisungsblocks werden sequentiell ausgeführt: Von der ersten bis zur letzten.

Bedingte Anweisung

Bedingte Anweisung

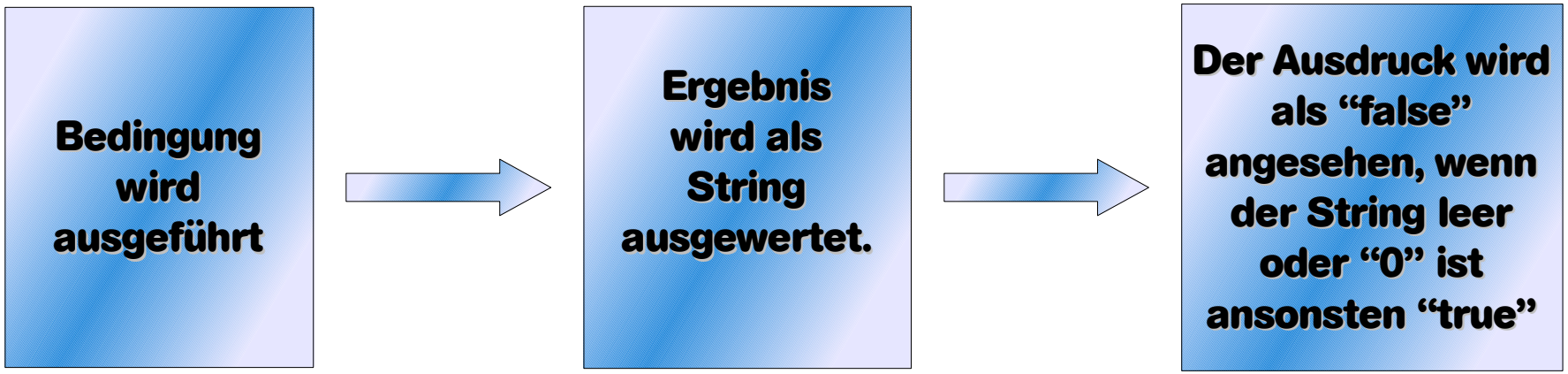


Die if/unless Anweisung

Zum if-Konstrukt gehört eine Bedingung und ein Block. Alternativ dazu kann es noch ein else mit einem entsprechenden Block geben.

```
if (some_expression) {
    true_statement_1;
    true_statement_2;
    true_statement_3;
} else {
    false_statement_1;
    false_statement_2;
    false_statement_3;
}
```

The Control Expression




```
0         false, weil 0 zu "0" konvertiert
3-3       Ergebnis der Subtraktion 0, dann
           Umwandlung in "0", daher “false”
""        leerer String, deshalb falsch
"1"       weder leer "" noch "0", daher “true”
1         konvertiert zu  "1", deshalb “true”
"00"      weder "" noch "0", deshalb true
"0.000"   ebenfalls true aus dem gleichen Grund
undef     wird als "" ausgewerte, deshalb false
```

Beispiel einer if-Anweisung


```
print "How old are you? ";
$age = <STDIN>;
chomp($age);
if ($age < 18) {
    print "Sorry, not old enough to vote!\n";
} else {
    print "Old enough! So go vote!\n";
    $possible_voter++;
}
```

if ohne else-part


```
print "How old are you? ";
$age = <STDIN>;
chomp($age);
if ($age < 18) {
    print "Sorry, not old enough to vote!\n";
}
```

unless

If you just want the else part and no then part, you can use the unless statement:


```
print "How old are you? ";
$age = <STDIN>;
chomp($age);
unless ($age < 18) {
    print "Old enough! So go vote!\n";
    $possible_voter++;
}
```

elsif for more than two choices

```
if (expression_1) {
    one_true_statement_1;
    one_true_statement_2;
    one_true_statement_3;
} elsif (expression_2) {
    two_true_statement_1;
    two_true_statement_2;
    two_true_statement_3;
} elsif (expression_3) {
    three_true_statement_1;
    three_true_statement_2;
    three_true_statement_3;
} else {
    all_false_statement_1;
    all_false_statement_2;
    all_false_statement_3;
}
```

Jeder Ausdruck (expression_1, expression_2 and expression_3) wird der Reihe nach berechnet. Wenn einer der Ausdrücke “true” ist, wird der entsprechende Ast ausgewertet und die anderen Kontrollstrukturen und die anderen Anweisungsblocks werden

Switch-Anweisung

Perl hat keine Switch-Anweisung

Lösung: Hashes!

Man benutzt subroutine-Referenzen Im HASH t

```
$action_to_take = (
        1 => \&process_direct_deposits,
        2 => \&query_account_status,
        3 => \&do_exit,
    );
```

```
$action_to_take{$menu_item}->();
```

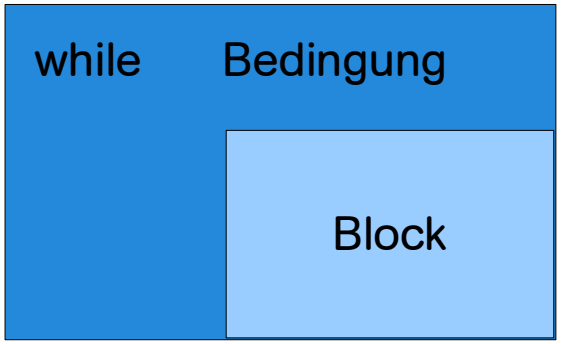
Schleifen

In den meisten Programmiersprachen dienen Schleifen dazu einen Anweisungsblock zu wiederholen, solange eine Bedingung erfüllt ist.

Es gibt drei Schleifentypen

- while
- until und
- for-Schleifen.

while Anweisung




```
while (expression){
    statement_1;
    statement_2;
    statement_3;
}
```

Perl wertet den Ausdruck `expression` aus. Ist er true wird der Anweisungsblock in den geschweiften Klammern einmal ausgeführt.

Dies wird solange wiederholt bis der der Kontrolausdruck fals ergibt. Dann macht Perl mit der ersten Anweisung nach der while-Schleife weiter.

`until` Anweisung

Wenn das Schlüsselwort `while` durch until ersetzt wird, kehrt sich der Test um, d.h. er wird negiert. Der Block wird dann solange ausgeführt bis der Ausdruck true ergibt.

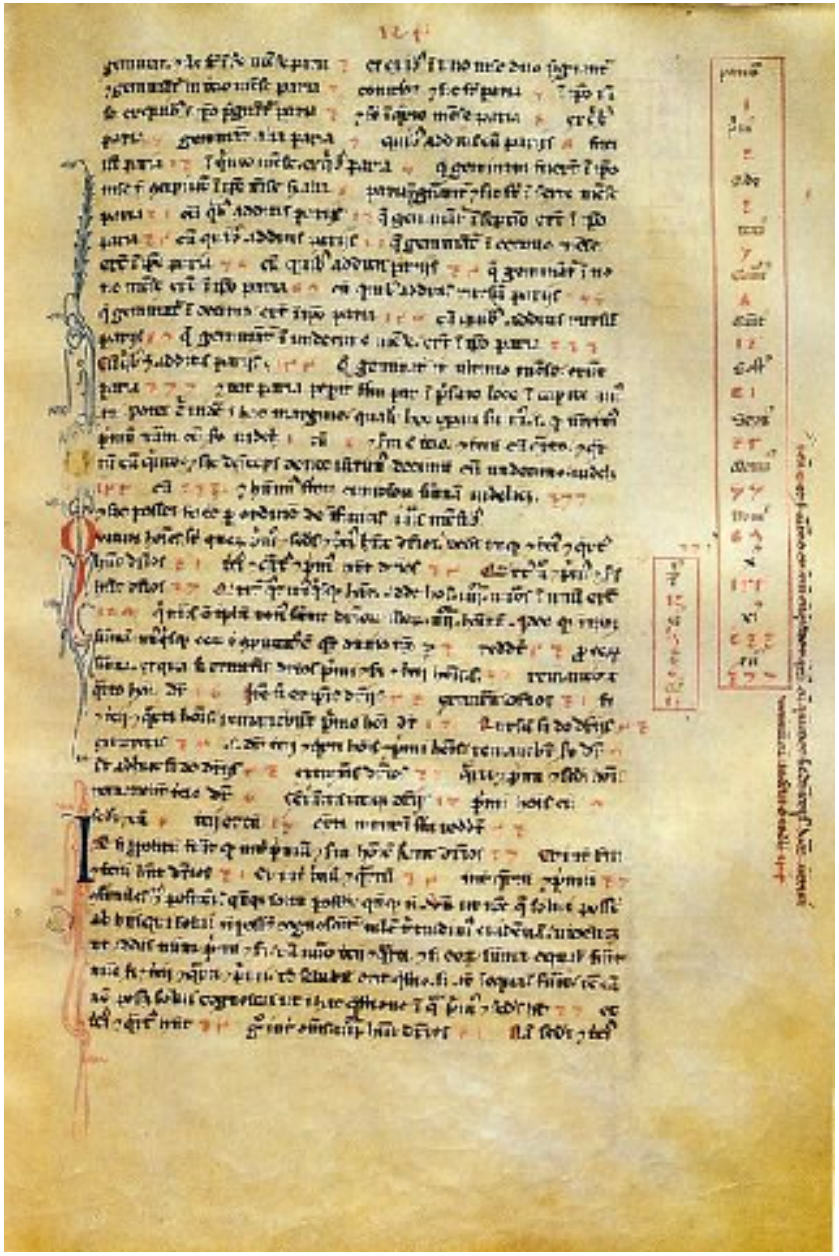
Auch in diesem Fall wird die Bedingung vor einer erstmaligen Ausführung des Blocks getestet.

Leonardi Fibonacci

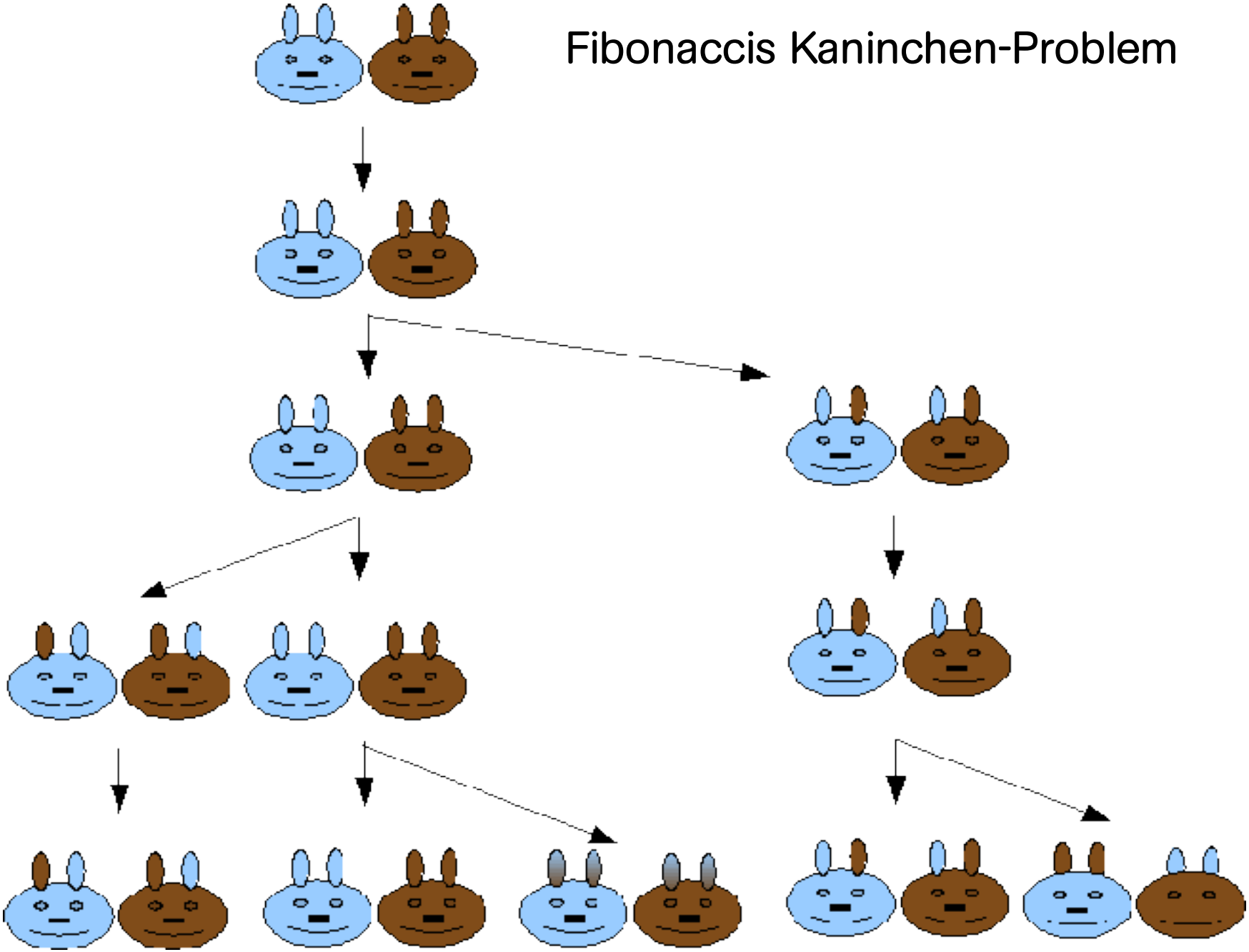
Leonardo da Pisa,
besser bekannt als
Fibonacci
figlio de Bonacio
(1180 - 1241)

einer der bedeutendsten
Mathematiker des
Mittelalters

sein bedeutendstes Werk:
Liber abbaci



Fibonaccis Kaninchen-Problem



Übung

Definition: Jedes Glied der Fibonacci-Folge ergibt sich aus der Summe der beiden Vorgänger. Die ersten sieben sind: 1, 1, 2, 3, 5, 8, and 13.

Formale Definition:

Die n-te Fibonacci-Zahl errechnet sich
F(n) = F(n-1) + F(n-2),
where F(1)=1 and F(2)=1

Aufgabe:
Schreibe ein Skript um F(n) zu berechnen!

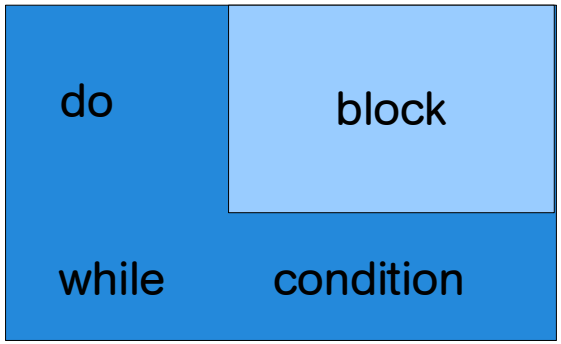
Lösung

```
#!/usr/bin/perl -w
print "Number? ";
$n = <STDIN>;
chomp($n);
if ($n >= 1) {
    $i = 1;
    $fib_n_1 = 0;
    $fib_n = 1;
    while ( $i < $n ) {
   $help = $fib_n + $fib_n_1;
   $fib_n_1 = $fib_n;
   $fib_n = $help;
   print "schnitt: " . $fib_n / $fib_n_1 ."\n";
   $i++;
    }
    print "Fibonacci number $n: $fib_n\n";
} else {
    print "Please start with a number >= 1\n";
}
```

do {} while/until Anweisung

```
do {
    statement_1;
    statement_2;
    statement_3;
} while expression;
```

Dieses Konstrukt ist dem vorher behandelten `while/until`ähnlich, allerdings wird der Block immer einmal ausgeführt und dann erst die Bedingung ausgewertet.



von while zu for

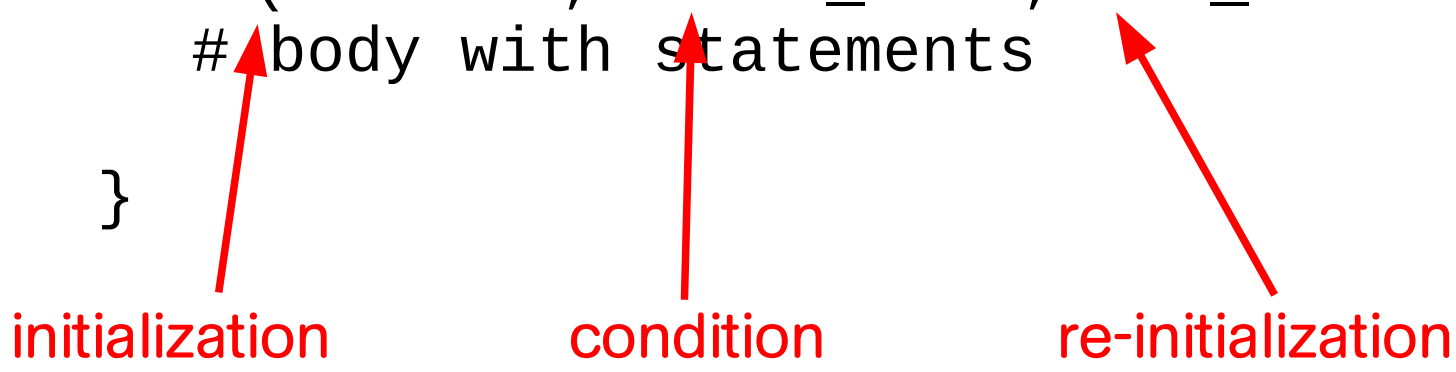
Arbeitet man ständig mit `while`-Schleifen, hat man meistens typische Strukturen wie die folgende:

```
# initialization of a loop variable
# affecting the test_exp:
$i = 0;
while (test_exp) {
   # a sequence of statements:
   statement_1;
   statement_2;
   ...
   # incrementing or decrementing the
   # loop variable
  $i++;
}
```

for-Anweisung

```
for (<INITS>; <TEST_EXP>; <RE_INITS> ) {
   #body with statements

}
```



<INITS> Eine durch Komma getrennte Liste von Schleifenvariablen-Initialisierungen.

<TEST_EXPR> Durchlaufbedingung die von den Variablen des Initialisierungsteils beeinflusst wird. <INITS>

<RE_INITS> Eine durch Komma getrennte Liste von Schleifenvariablen-Zuweisungen.

Beispiel

Das folgende Skript druckt die Quadrate von 1 bis 100.

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

for ($i=1; $i < 100; $i++) {
    print "$i: " . $i**2 . "\n";
}
```

Endlosschleifen mit `while` und `for`

```
for (;;) {

}

while (1) {

}
```

`foreach` Loops

The foreach-Schleife iteriert über eine Liste von Variablen, indem bei jedem Durchgang die Kontrollvariable auf das jeweils nächste Element der Liste gesetzt wird.

```
foreach $VAR (LIST) {
    ...
}
```

Statt dem `foreach` Schlüsselwort kann man auch `for` benutzen, denn `foreach` ist nur ein Synonym für `for`.

Beispiel

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

@l = (3, 4, 6, 8);
foreach $i (@l) {
    print $i**2 . "\n";
}
```

Eine kleine Änderung mit großen Folgen:

```
@l = (3, 4, 6, 8);
foreach $i (@l) {
    $i = $i**2 . "\n";
    print $i;
}
```

`@l` änderte sich zu `(9, 16, 36, 64)`

Labelling Loops

Einer Schleife kann ein Label vorangestellt werden. Dieses Label identifiziert den Schleifenanfang für die folgenden Schleifenoperatoren:

- next
- last
- redo

Wenn ein Loop-Operator mit einem Label benutzt wird, wird dieses Label benutzt ansonsten wird die innerste Schleife als Ansprungziel benutzt.

next

```
next [LABEL]
```

Das next Kommando ist wie die continue Anweisung von C:

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

Falls ein continue-Block existiert, wird dieser ausgeführt. Danach wird entweder beim Anfang der innersten Schleife weitergemacht oder, wenn ein Label angeben wurde, beim Label weitergemacht.

Beispiel with next

```
$n = <STDIN>; # the final number
print "$n\n";
$count = 0;
while ($count <= $n) {
   # set a conditional statement to interrupt
   if (( ($count % 4) == 0) && ($count != 0)) {
      print "$count is divisible by 4!\n";
      $count ++;
      next;
   }
   # go on as usual
   print $count."\n";
   $count ++;
}
   continue {
      print $count."\n";
      $count++;
   };
print "That's all! Task finished!";
```

last

```
last [LABEL]
```

Das `last`-Kommando entspricht der break-Anweisung von C; durch dieses Kommando wird die Schleife sofort verlassen.

Wird kein Label angegeben wird die innerste Schleife verlassen ansonsten die durch das Label bezeichnete.

Ein continue-Block wird nicht ausgeführt auch wenn er vorhanden ist:

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

Beispiel mit next und last

```
my $count = 0;
LINE: while (<STDIN>) {
    next LINE if /^#/;  # skip comment lines
    next LINE if /^$/;  # skip blank lines
    print;   # $_
    last LINE if /eee/;
} continue {
    $count++;
}
print "$count lines have been input!\n";
```

redo

```
redo [LABEL]
```

Das redo-Kommando startet den Block wieder von Anfang an, ohne die Bedingung auszuführen. Ein eventuell vorhandener continue-Block wird nicht ausgeführt.

Auch hier bezieht sich das Kommando auf die innerste Schleife, wenn kein Label angegeben worden ist.

Beispiel

```
# bad luck, an infinite loop:
$counter = 1;
while ($counter < 20) {
    redo if ($counter == 13);
    print "counter: $counter\n";
}
continue {
     $counter++;
};
```

Blocks als Schleifen

Ein einfacher Block unabhängig davon, ob er benannt (labeled) ist oder nicht, ist semantisch gesehen äquivalent mit einer Schleife, die nur einmal ausgeführt wird.

Ausnahmen:

- Gilt nicht für die Blöcke von eval {}, sub {} and do {}.
- funktionieren auch nicht mit `if` und `unless`.

Trick für do {}

Indem man einen zuätzlichen einfachen Block innerhalb der do-Klammern einfügt, kann man `next` und `redo` dennoch benutzen:

```
do {{
    next if .....;
    # some statements following
}} until ....;
```

So kann man auch `last` benutzen:

```
{
    do {
        last if ...;
        # some statements following
    } while ...;
}
```

do {} Trick, Fortsetzung

Um sowohl next als auch last in der do {} Anweisung zu benutzen brauchen wir zusätzlich noch Labels:

```
DO_LAST: {
   do {
DO_NEXT:  {
      next DO_NEXT if ...;
      last DO_LAST if ...;
     # some statements following
           }
      } while ...;
}
```

Case und Switch in Perl

Perl hat keine case oder switch wie die meisten anderen Programmiersprachen.

Aber dieses Verhalten kann man einfach mit einfachen Blöcken simulieren:

```
SWITCH: {
    if (...) { $abc = 1; last SWITCH; }
    if (/.../) { $def = 1; last SWITCH; }
    if (/.../) { $xyz = 1; last SWITCH; }
    $nothing = 1;
}
```

Case und Switch in Perl, Forts.

eine alternative Formulierung:

```
SWITCH: {
    /^abc/      && do { $abc = 1; last SWITCH; };
    /^def/      && do { $def = 1; last SWITCH; };
    /^xyz/      && do { $xyz = 1; last SWITCH; };
    $nothing = 1;
}
```

Subroutinen

Eine Subroutine ist ein Unterprogramm oder eine Funktion die in einem Perl-Skript wie folgt definiert wird:

```
sub subname {
    statement_1;
    statement_2;
    statement_3;
}
```

Der Name einer Subroutine (`subname`) wird formuliert wie die Variablennamen.

Die Funktionsnamen erscheinen in einem andern Namensraum als eine skalare Variable `$foo`, ein Array `@foo`, ein Hash `%foo`.

Subroutine Beispiel

```
sub hello_you_know_who {
   $number_of_times_called++;
   print “Hello World!\n”;
}
```

Aufgerufen wird diese Subroutine mit:

```
hello_you_know_who();
```

Return Values

Der Wert, den eine Subroutine bei ihrem Aufruf zurückliefert, wird als Rückgabewert der Funktion (return value) bezeichnet.

Der Rückgabewert einer subroutine entspricht dem Wert der letzten return-Anweisung oder des letzten Ausdrucks der in der Subroutine ausgeführt worden ist.

```
sub areaOfCircle {
    $radius = $_[0];
    return(3.1415 * ($radius ** 2));
}
```

Positionierung der Subroutinen

Subroutinen-Definitionen können überall im Skript positioniert werden.

Man sollte sie jedoch am besten einheitlich an einem Ort positionieren, d.h. am Anfang (C-Stil) oder am Ende (Perl-Stil).

Subroutinen-Definitionen sind immer global, d.h. es gibt keine globale Subroutine.

Falls zwei Subroutinen mit dem gleichen Namen existieren, überschreibt die zuletzt definierte die erste.

Wenn nicht anders bezeichnet, sind die Variablen innerhalb einer Subroutine global.

Arguments

Ein Subroutinen-Aufruf wird von einer (möglicherweise leeren) Liste von Argumenten gefolgt.

Die Werte dieser Liste werden automatisch für die Dauer der Subroutine einer Array-Variablen @_ zugeordnet.

```
sub ListSum {
    $sum = 0;           # initialize the sum
    foreach $x (@_) {
        $sum += $x;     # add element
    }
return  $sum;           # sum of all elements
}

print ListSum(1,2,3,4,5,6);
```

Globale und lokale Variablen

Subroutinen: Lokale Variablen

Die @_ Variable ist local in der Subroutine,
ebenso wie $_[0], $_[1], $_[2],

Andere lokale Variables können wie folgt deklariert werden

```
local(<comma separated list of variable names>);
```

Der my Operator nimmt eine Liste von Variablennamen und schafft lokale Versionen dieser Variablen.

Aufruf einer Benutzer-Funktion

```
sub max {
    if ($_[0] > $_[1]) {
   $_[0];
    } else {
   $_[1];
    }
}
print("enter first number: ");
$number1 = <STDIN>;
chomp($number1);
print("enter second number: ");
$number2 = <STDIN>;
chomp($number2);

$max = max($number1, $number2);
print "max: $max\n";
```

Unterschied my und local

```
sub PrintX {
    print "x: $x\n";
}

sub UsingLocal {
    local($x) = 5;
    PrintX();
}

sub UsingMy {
    my($x) = 5;
    PrintX();
}

$x = 1;
PrintX();
UsingLocal();

$x = 1;
PrintX();
UsingMy();
```

Variablen, die mit `local` deklariert wurden sind innerhalb der Funktion gültig, in der sie definiert wurden und in jeder Funktion, die innerhalb dieser Funktion aufgerufen werden.

Variablen, die mit `my` deklariert wurden sind nur innerhalb der Funktion gültig, in der sie deklariert wurden.

Noch ein Unterschied

`my` kann nur dazu genutzt werden eine Skalare, Array- oder Hash-Variablen zu deklarieren. Für `local` gibt es keine Restriktionen.

Perl‘s built-in Variablen, wie zum Beispiel `$_`, `$1`, und `@ARGV` können nicht mit `my` deklariert werden, aber mit `local`.

Kommandozeilen-Argumente

Perl speichert Kommandozeilen-Argumente in einem Array @ARGV, d.h. $ARGV[0] enthält das erste Argument $ARGV[1] enthält das zweite und so weiter.

```
#!/usr/bin/perl

$numArgs = $#ARGV + 1;
print "$numArgs command-line arguments.\n";
print "The arguments are:\n";
foreach $argnum (0 .. $#ARGV) {
   print "$ARGV[$argnum]\n";
}
```

Übung: Ackermann Funktion

Schreibe Sie ein Programm das eine rekursive Funtion benutzt um die Ackermann-funktion zu berechnen:

$$A(m,n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m-1,\ 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1,\ A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Ackermann-Funktion in Perl

```
sub A {
    local($x,$y,$x_1, $y_1);
    $x = $_[0];
    $y = $_[1];
    if ($x == 0) {
        $y+1;
    } elsif ($y == 0) {
        $x_1 = $x - 1;
        A($x_1, 1);
    } else {
        $x_1 = $x - 1;
        $y_1 = $y - 1;
        A($x_1, A($x,$y_1));
    }
}
```

Ackermann Funktion: Resultate

n \ m	0	1	2	3	4	m
0	1	2	3	4	5	$m+1$
1	2	3	4	5	6	$m+2$
2	3	5	7	9	11	$2m+3$
3	5	13	29	61	125	$8 \cdot 2^m - 3$
4	13	65533	$2^{65536} - 3 \approx 2 \cdot 10^{19728}$ [1]	$a(3,2^{65536}-3)$	$a(3,a(4,3))$	$2^{2^{\cdot^{\cdot^{2}}}} - 3$ ($m+3$ Terme)
5	65533	$a(4,65533)$	$a(4,a(5,1))$	$a(4,a(5,2))$	$a(4,a(5,3))$	
6	$a(5,1)$	$a(5,a(5,1))$	$a(5,a(6,1))$	$a(5,a(6,2))$	$a(5,a(6,3))$	

Hintergründe

Addition:

$$a + b = a + \underbrace{1 + 1 + \ldots + 1}_{b}$$

Multiplikation

$$a \times b = \underbrace{a + a + \ldots + a}_{b}$$

Exponentiation

$$a^b = \underbrace{a \times a \times \ldots \times a}_{b}$$

Tetration (hyper-4)

$${}^b a = \underbrace{a^{a^{\cdot^{\cdot^{a}}}}}_{b}$$

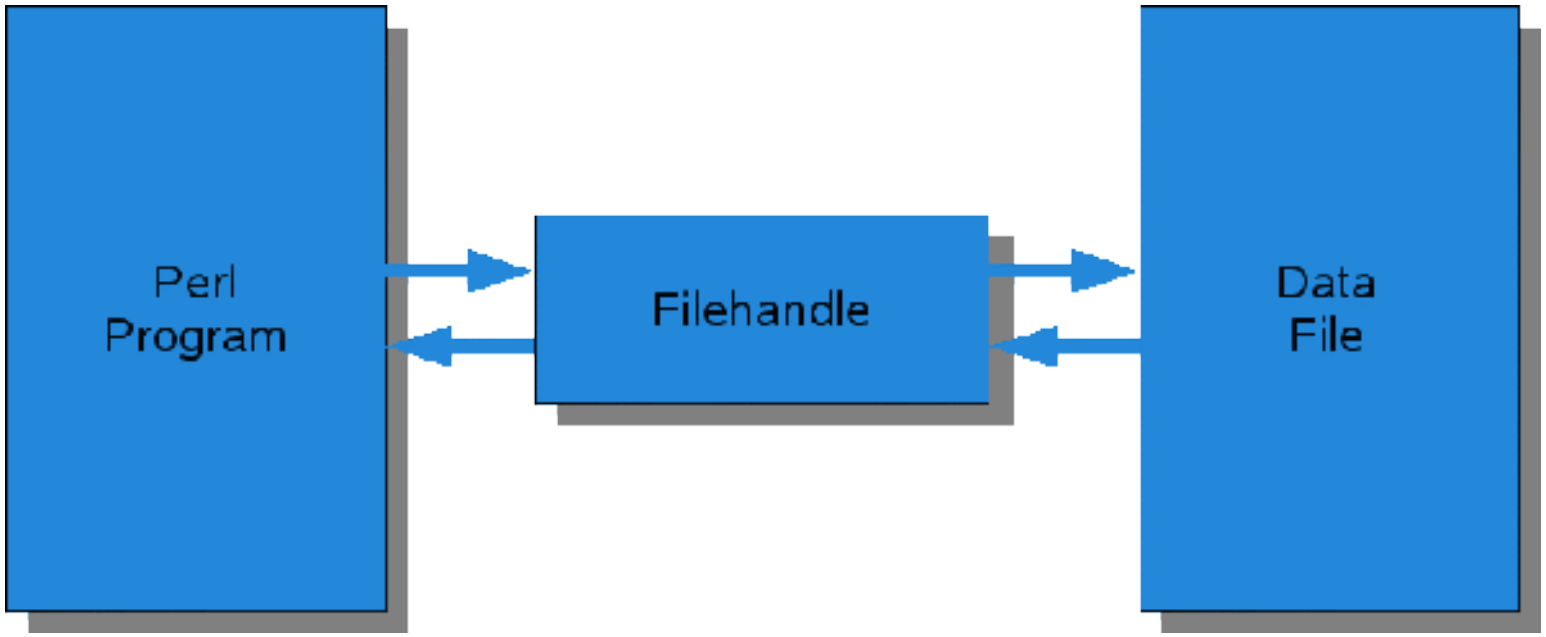
Filehandles und File Tests

Ein Filehandle in einem Perl-Programm ist eine Bezeichnung (Name) für eine I/O-Verbindung zwischen einem Perl-Prozess und der Außenwelt.

STDIN ist ein Filehandle, der die Verbindung zwischen dem Perl-Prozess und dem Standard-Input bezeichnet. STDOUT (Standard Output) und STDERR (Standard Error Output) sind weiter Filehandels.

It‘s customary to CAPITALIZE the names of filehandles.

Was ist ein Filehandle



In anderen Worten: Ein Filehandle kann als ein weiterer Name für die Dateien aufgefasst werden, die im Perl-Skript benutzt werden.

Ein Filehandle ist ein temporärer File, der einer Datei zugeordnet wird. Üblicherweise benutzt man eine abgekürzte Version des Dateinamens.

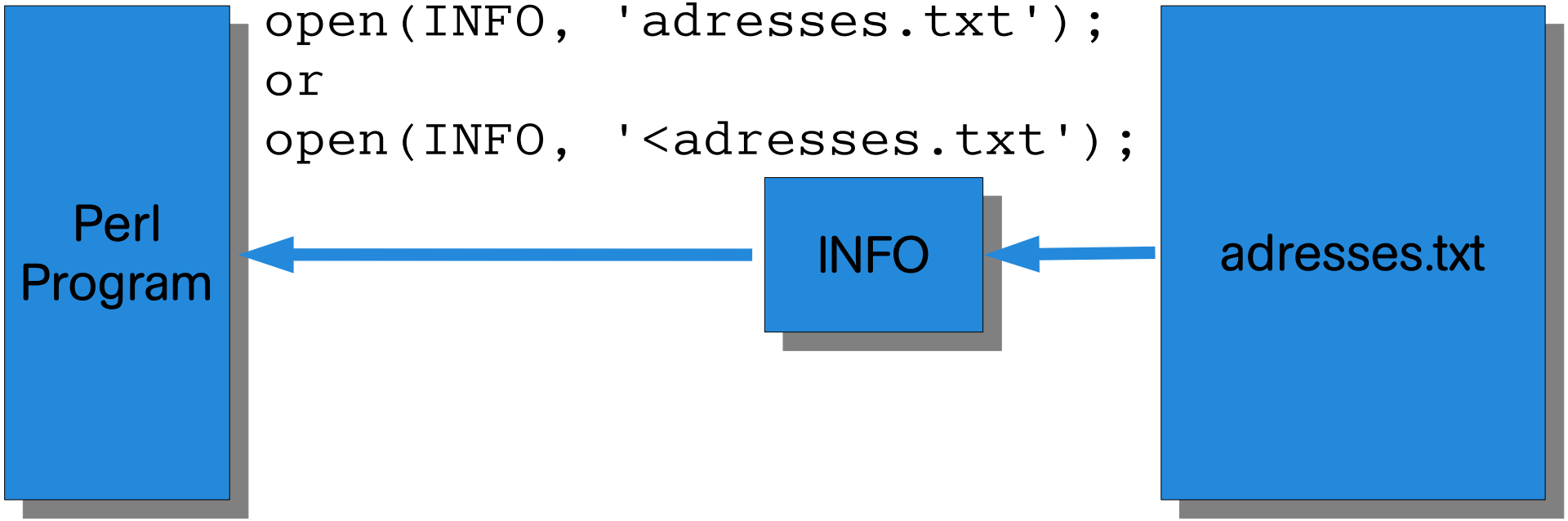
Opening und Closing

Filehandles müssen mit der open()-Anweisung geöffnet (open) werden, außer bei STDIN, STDOUT und STDERR, die immer automatisch geöffnet sind.

```
$file = '/home/homer/addresses.txt';
open(INFO, $file);          # Open the file
```

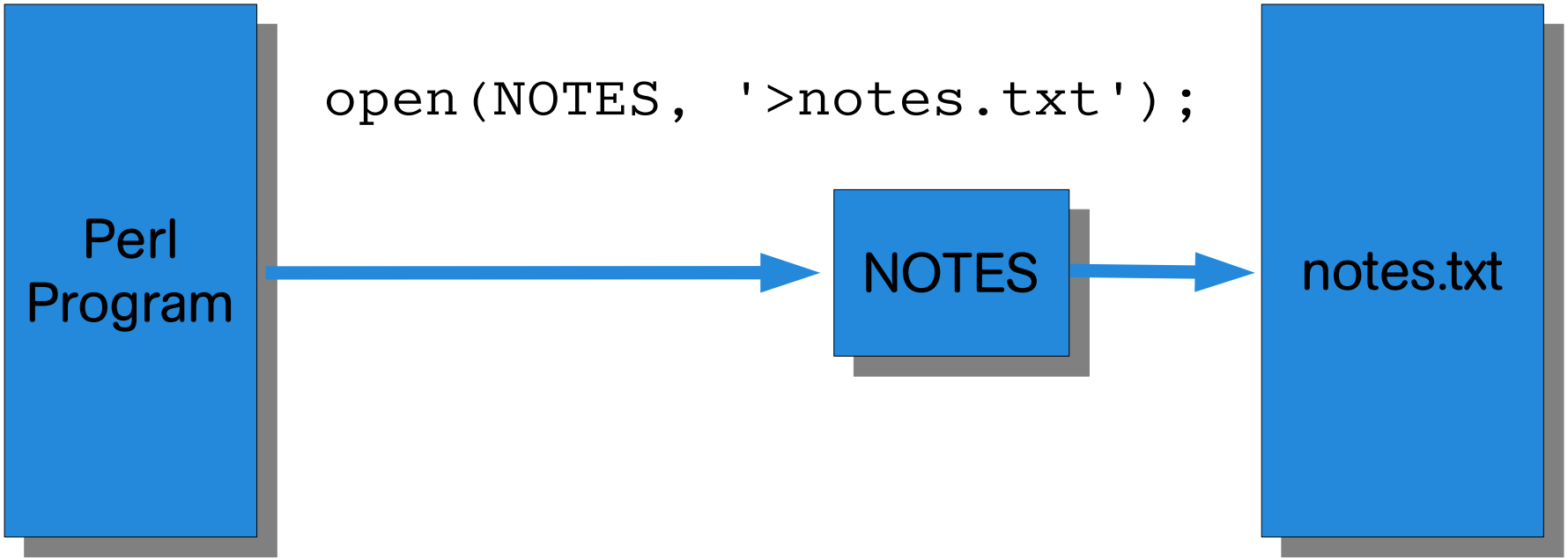
Zum Lesen öffnen

Eine Datei wird zum Lesen geöffnet:



Zum Schreiben öffnen

Eine Datei wird zum Schreiben geöffnet:

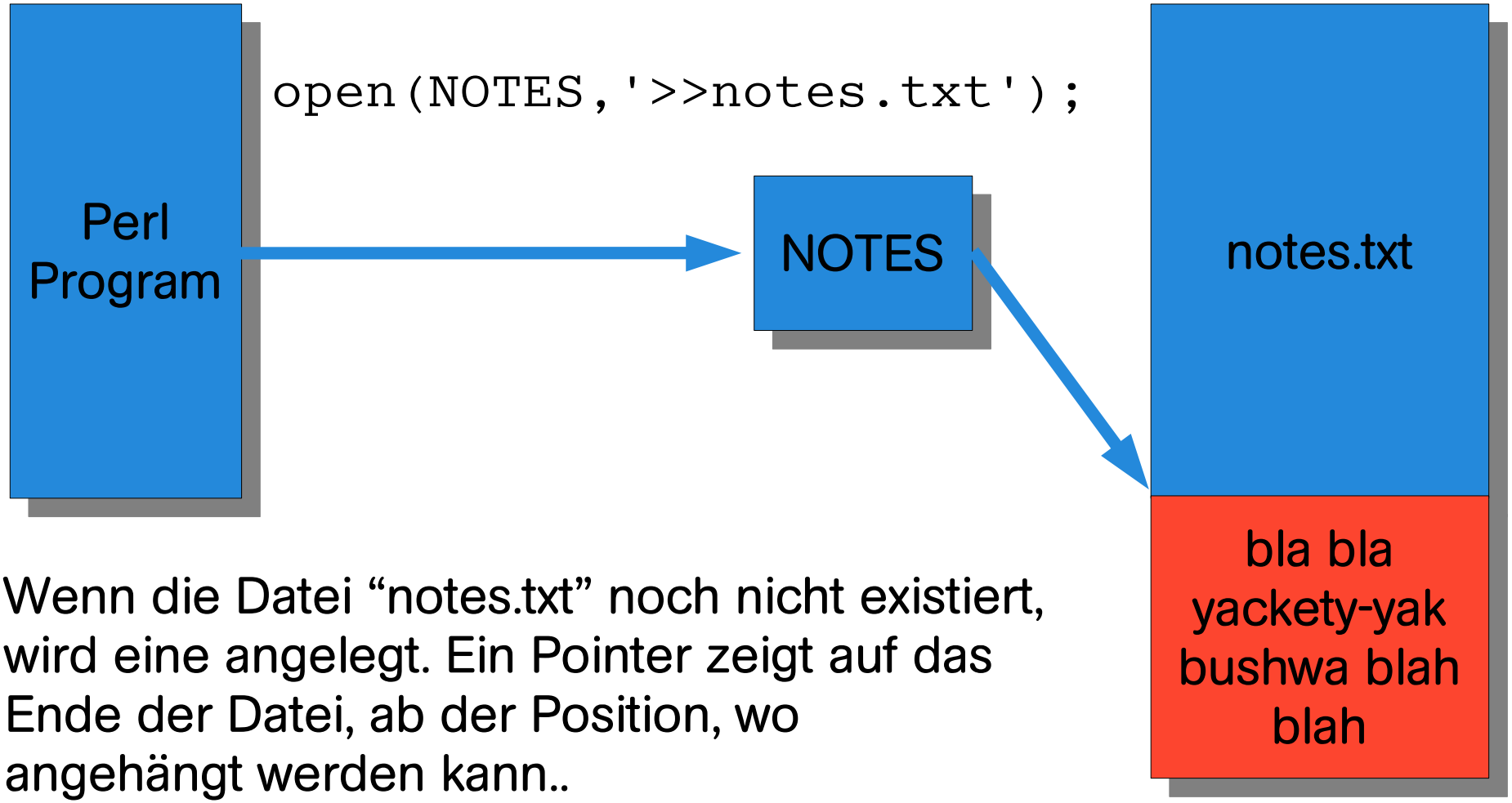


Wenn die Datei “notes.txt” bereits existiert, wird sie überschrieben

Die Daten gehen dabei verloren!

Zum Schreiben mit Anhängen öffnen

An eine (existierende) Datei anhängen:



Wenn die Datei “notes.txt” noch nicht existiert, wird eine angelegt. Ein Pointer zeigt auf das Ende der Datei, ab der Position, wo angehängt werden kann..

Bizarre Dinge über open

Klammern können weggelassen werden:

```
open INFO, $file;
```

Der Dateiname kann auch weggelassen werden, wenn eine skalare Variable mit dem gleichen Namen wie der Name des Filehandles existiert:

```
$INFO = '<addresses.txt';
open INFO ;
@lines = <INFO>;
close(INFO);
print @lines;
```

Problem mit open

open liefert true für **success** und false im Falle eine **Fehlschlags** zurück.

Gründe für ein Versagen:

Beim lesenden Zugriff auf eine Datei, gibt es z.B. folgende Fehlermöglichkeiten:
Die Datei existiert nicht oder
kann wegen wegen mangelnder Zugriffsrechte nicht geöffnet werden.

Beim schreibenden Zugriff kann es folgende Probleme geben:

Datei ist schreibgeschützt oder

auf das Verzeichnis darf nicht zugegriffen werden.

Der `close`-Operator

Wenn ein Skript den Zugriff auf eine Datei nicht mehr benötigt, kann man die Datei mit dem `close`-Operator schließen:

```
close(FILEHANDLE);
```

Ein erneutes Öffnen eines Filehandles schließt automatisch die noch geöffnete Datei.

open für Spezialisten

Bis jetzt haben wir uns “open à la C” angeschaut.

sysopen ist das Kommando für feinere Abstufungen.

```
sysopen HANDLE, PATH, FLAGS, [MASK]
```

Abfangen von open-Problemen

Konnte ein Filehandle nicht geöffnet werden, kann es dennoch innerhalb des Programmes benutzt werden, ohne dass man eine Warnung erhält.

Perl verhält sich so, als hätte man im Falle des lesenden Zugriffs bereits das Ende der Datei erreicht oder im Fall des schreibenden Zugriffs wird das zu Schreibende einfach “kommentarlos weggeworfen”.

Hässliche Lösung:

```
unless (open (INFO,">addresses.txt")) {
    print "I couldn't create addresses.txt\n";
} else {
    # the rest of your program
}
```

Die perfekte Lösung

die

Die die-Funktion nimmt eine Liste innerhalb von optionalen Klammern und gibt diese Liste auf dem Standard-Error-Output aus. Danach wird das Perl-Programm mit einem Status ungleich Null beendet.

```
unless (open (INFO,">addresses.txt")) {
    die "I couldn't create addresses.txt\n";
}
# the rest of the program
```

Open die Datei oder die

```
open (INFO,">addresses.txt")||
    die "I couldn't create addresses.txt\n";

# the rest of the program
```

|| (logical-or) stellt sicher, dass das Kommando die nur ausgeführt wird, wenn open nicht geklappt hat.

Zeilen von einer Datei einlesen

Ist ein Filehandle zum Lesen geöffnet, kann man daraus wie vom Standard Input lesen.

```
$file = '/home/homer/addresses.txt';
open(INFO, $file);          # Open the file
@lines = <INFO>;            # Read it into an array
close(INFO);                # Close the file
print @lines;               # Print the array
```

alternatives Vorgehen:

```
open (INFO,"/home/homer/addresses.txt");
while (<INFO>) {
    chomp;
    print "$_\n";
}
```

In eine Datei schreiben

Ist ein Filehande zum Schreiben geöffnet, kann man in die Datei mit print schreiben. Allerdings muss nach dem print der Name des Filehandles folgen:

```
print RESULTS "The value is: $n\n";
```

Programm: Dateien kopieren

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

$in_file = "addresses.txt";
$out_file = ">addresses2.txt";

open(IN,$in_file) || die "cannot open $in_file
for reading: $!";
open(OUT,$out_file) || die "cannot create
$out_file: $!";
while (<IN>) {      # read a line from file $a
into $_
    print OUT $_;   # print that line to file $b
}
close(IN) || die "can't close  $in_file: $!";
close(OUT) || die "can't close  $out_file: $!";
```

nur um auf der sicheren Seite zu sein ...

Eine bereits existierende Datei ist schnell überschrieben.
Deshalb ist es sinnvoll vorher zu schauen, ob eine solche Datei bereits existiert.

Dazu gibt es in Perl ein Kommando. Mit `-e $filevar` kann man prüfen, ob eine Datei bereits existiert.

```
$in_file = "addresses.txt";
if (-e $in_file) {
    print "The file $in_file already exists";
} else {
    print "The file $in_file doesn't exist";
}
```

Tests auf Dateien u. Verzeichnissen

```
-r  File or directory is readable
-w  File or directory is writable
-x  File or directory is executable
-o  File or directory is owned by user
-e  File or directory exists
-z  File exists and has zero size (directories are never empty)
-s  File or directory exists and has nonzero size
    (the value is the size in bytes)
-f  Entry is a plain file
-d  Entry is a directory
-l  Entry is a symlink
-S  Entry is a socket
-p  Entry is a named pipe (a "fifo")
-b  Entry is a block-special file (like a mountable disk)
-c  Entry is a character-special file (like an I/O device)
-k  File or directory has the sticky bit set
-T  File is "text"
-B  File is "binary"
-M  Modification age in days
-A  Access age in days
-C  Inode-modification age in days
```

Alles über eine Datei mit `stat`

Sie möchten alles über eine Datei erfahren?

```
# check if STDIN is interactive and prompt if it is
print "File name? " if (-t STDIN);
chop ($name = <STDIN>);

@file_data = stat($name);
@description =("device", "inode", "mode", "links",
"user id", "group id", "device id", "size",
"accessed", "modified", "changed", "blocksize",
"block count");

foreach $index (0 .. $#description) {
    printf "%-12s",$description[$index];
    print $file_data[$index],"\n";
}
```

Unterschied: stat() und lstat()

Falls es sich bei einem Argument nicht um einen symbolischen Link handelt liefern `stat()` und `lstat()` das selbe Resultat.

Ruft man die `stat()`-Funktion jedoch mit einem symbolischen Link auf, erhält man nur Informationen über die Datei, auf die der Link zeigt und nicht über den symbolischen Link selbst.

`lstat()` liefert im letzten Fall jedoch Informationen über den symbolischen Link selbst.

Übungen

1) Schreiben Sie ein Programm, dass eine Datei einliest und dann in eine andere Datei schreibt, aber jede Zeile soll mit einer fortlaufenden Zeilennummer versehen sein.

2) Schreibe Sie ein Programm, dass eine Liste von Dateinamen einliest und dann ausgibt welche lesbar, schreibar und/oder ausführbar sind.

Mehrere Dateien einlesen

Falls innerhalb des Diamanten-Operators kein Argument angegeben ist, wird von Perl die @ARGV-Variable getestet. Falls @ARGV leer ist, wird von `STDIN` gelesen, d.h. von der Tastatur oder aus einer umgelenkten Datei.

```
while (<>) {
    print();
}
```

Wird der Diamanten-Operator wie folgt aufgerufen

```
multiple_file_read.pl abc.txt efg.txt
```

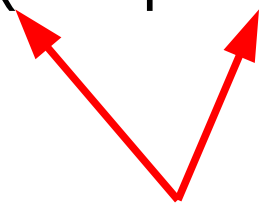
wird der Inhalt der Datei `abc.txt` gefolgt von `efg.txt` ausgedruckt.

Zugriff auf ein Verzeichnis

Die `chdir`-Funktion benötigt ein Argument - das ausgewertete Argument wird als Verzeichnis-Pfad aufgefasst zu dem verzweigt werden soll.

`chdir` liefert `true` wenn das Skript in der Lage war in das Verzeichnis zu wechseln; ansonsten wird `false` zurückgeliefert.

```
chdir("/opt") || die "cannot cd to /opt ($!)";
```



Die Klammern sind optional

Anwendung von chdir

```
if ( chdir “/opt”) {
   print “We got there!”;
} else {
   print “We go to tmp instead!”;
   chdir /tmp;
}
```

Globbing

Die Expansion von *-Ausdrücken, wie z.B. `*` or `/home/homer/don*`, in eine Liste von passenden (matching) Dateinamen wird als **globbing** bezeichnet.

Um Globbing aufzurufen muss das Pattern zwischen spitzen Klammern stehen oder muss als Argument der `glob`-Funktion übergeben werden.

```
@a = </opt/kde*>;
@a = glob("/etc/*ca*");
```

Directory/Verzeichnis Handles

```
opendir DIRHANDLE,EXPR
```

`opendir` öffnet ein Verzeichnis mit dem Namen EXPR für die Verarbeitung als ”readdir”, ”telldir”, ”seekdir”, ”rewinddir”, and ”closedir”.
It returns `true` if successful. `DIRHANDLE` may be an expression whose value can be used as an indirect dirhandle, usually the real dirhandle name.

Dirhandless have their own namespace separate from Filehandles.

Lesen eines Directory Handle

Ist ein Verzeichnis-Handle geöffnet, können wir die im Verzeichnis enthaltenen Dateinamen als Liste einlesen.

Jeder Aufruf von `readdir` in einem skalaren Kontext liefert einen weiteren Dateinamen (ohne Pfad) in beliebiger Reihenfolge zurück.

Gibt es keine weiteren Dateinamen liefert `readdir undef zurück.`

Wird `readdir` innerhalb eines Listenkontextes aufgerufen, liefert es alle Dateinamen in einer Liste zurück.

readdir Beispiel

```
my $dir = '.';
opendir(DIR, $dir) or die $!;

while (my $file = readdir(DIR)) {
      # Use a regular expression to ignore
      # files beginning with a #
     next if ($file =~ /^[#]/);
     print "$file\n";
}

closedir(DIR);
exit 0;
```

Datei löschen

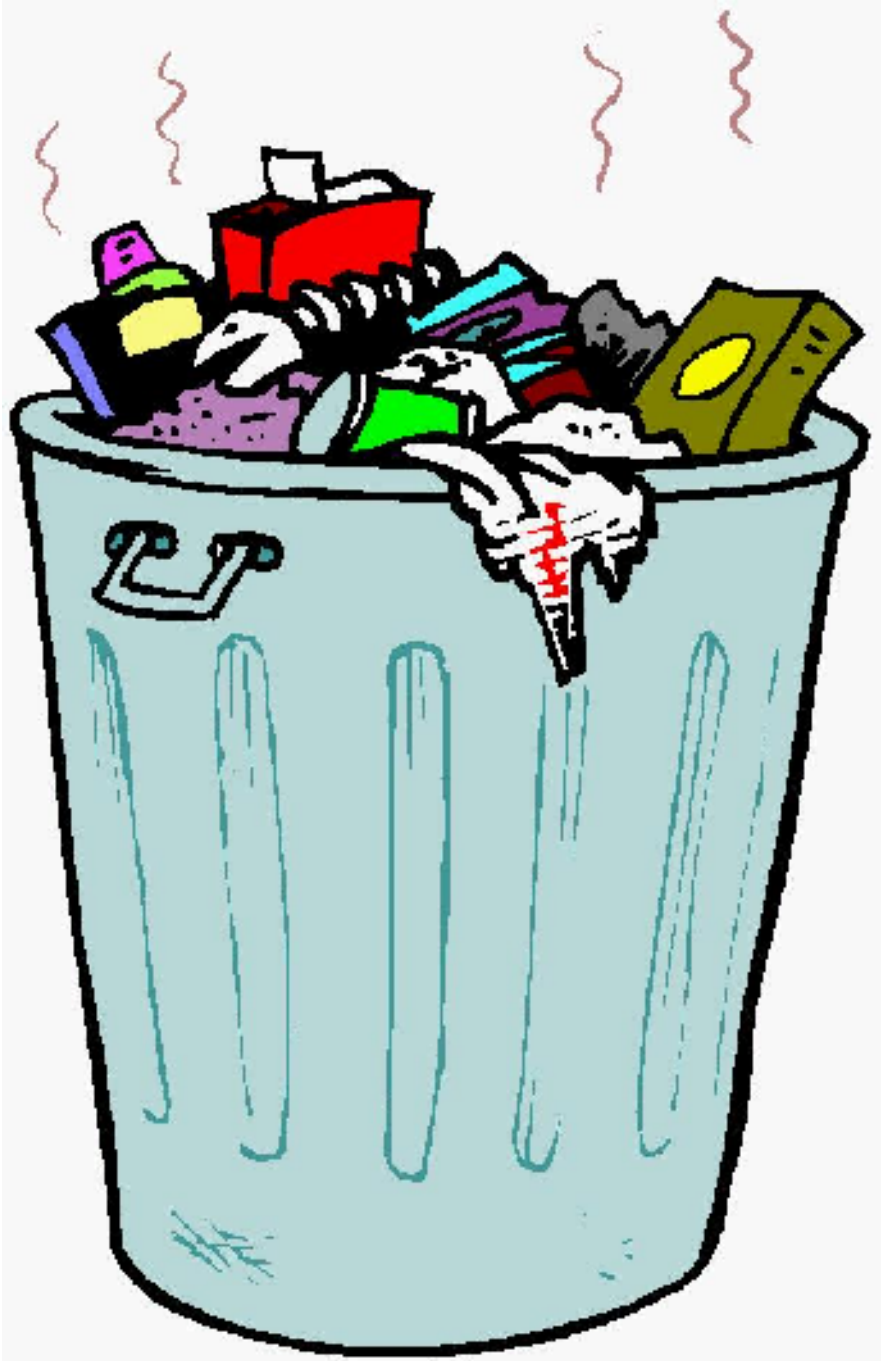
Das Perl-Kommando `unlink` löscht **einen** Namen einer Datei.

Falls es für eine Datei – was normalerweise der Fall ist – nur einen Namen gibt, wird der Name und die Datei selbst gelöscht.

```
unlink (“test.pl”);
```

Der `unlink`-Funktion kann man auch eine Liste von Dateinamen übergeben:

```
unlink (“a.pl”, “b.pl”);
unlink <*~>;
```



Rückgabewert von `unlink`

Der Rückgabewert von unlink entspricht der Anzahl der erfolgreich gelöschten Dateien.

Falls diese Zahl mit der Zahl der Dateinamen in der Argumentliste ist, gibt es keine Probleme. Falls die Anzahl kleiner ist, stellt sich die Frage, welche Dateien nicht gelöscht werden konnten.

Deshalb ist es meistens doch besser Dateien einzeln in einer Schleife zu löschen:

```
foreach $file (<*~>)
    unlink($file) || warn "having trouble deleting
$file: $!";
}
```

Umbenennen

Häufig möchte man auch gerne Dateinamen umbenennen.

Das ist sehr einfach in Perl:

```
rename($old_file, $new_file);
```

Natürlich sollte auch diese Operation auf ein Scheitern überwacht werden:

```
rename(Raider, Twix) || die “Renaming failed”
```

rename: Unterschied zu (UNIX) mv

In Unix/Linux sind die folgenden Kommandos äquivalent:

```
mv twix /opt/comp/                    and
mv twix /opt/comp/twix
```

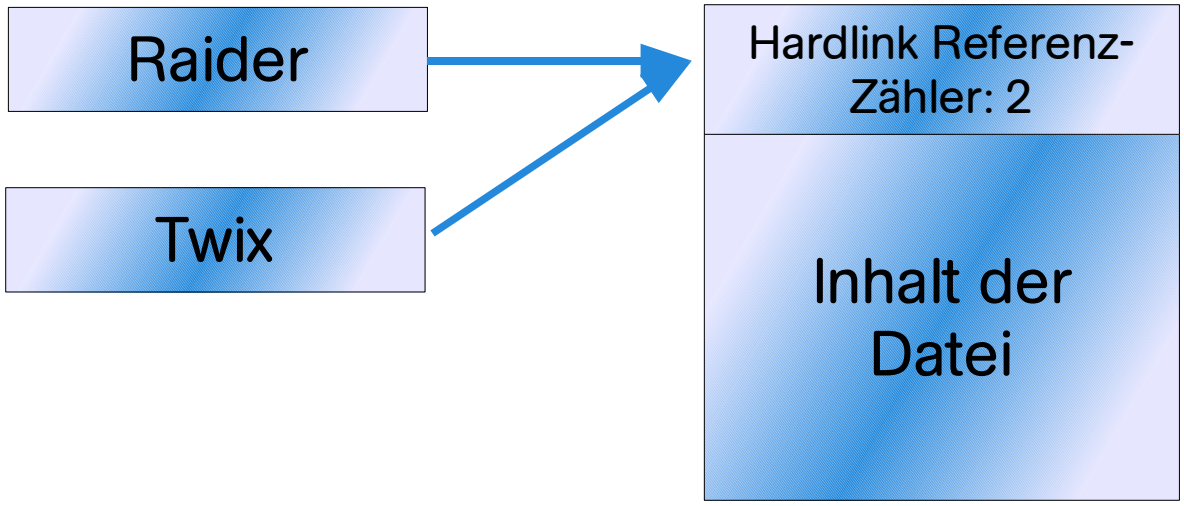
Aber in Perl muss der Ziel-Dateiname immer explizit angegeben werden:

```
rename(“twix”,”/opt/comp/twix”);
```

Hard und Soft-Links

Ein Hardlink ist ununterscheidbar von der Originaldatei.

Die Referenzen auf eine Datei werden hochgezählt



Einschränkungen für Hardlinks

Ein Hardlink muss sich auf der gleichen Partition wie die gelinkte Datei befinden.

Hardlinks auf ein Verzeichnis sind nicht möglich, weil das Dateisystem strikt hierarchisch organisiert ist.
Hardlinks auf Verzeichnisse würde zu zyklischen Graphen führen.

Symbolische Links

Sie werden auch “Soft-Links” genannt.

Wird ein symbolischer Link in einem Perl-Kommando benutzt, wird die verlinkte Datei anstelle des Links benutzt.

Ein symbolischer Link ist eine spezielle Datei, die einen Dateinamen (inklusive Pfad) als Daten enthält.

Der Inhalt von symbolischen Links muss nicht notwendigerweise auf existierende Dateien zeigen.

Ketten von symbolischen Links sind möglich.

Hard und Soft Links erzeugen

```
link($old_filename, $new_filename)
```

Ein Hardlink von `$new_filename` auf `$old_filename`.
Achtung: `$old_filename` muss existieren!

Ein Beispiel:

```
link(“raider”, “twix”)
     || die “cannot link raider to twix”;
```

Symbolische Links werden mit dem `symlink`-Kommando erzeugt:

```
symlink(“raider”, “twix”)
     || die “cannot link raider to twix”;
```

Bemerkung: “raider” muss nicht existieren and “twix” kann auf einer anderen Partition stehen.

Readlink

```
readlink( EXPR)
readlink EXPR
```

Liefert den Wert eines symbolischen Link zurück, wenn symbolische Links implementiert sind. Wenn nicht gibt es einen fatalen Fehler. Falls es einen System-Fehler gibt, liefert readlink den “undefinierten Wert” zurück und $! (errno) wird gesetzt.

```
if (defined($x = readlink("twix"))) {
    print "twix points at '$x'\n";
}
```

Verzeichnisse erzeugen

`mkdir(DIRNAME,MODE)`

Erzeugt ein Verzeichnis `DIRNAME` mit den in `MODE` angegebenen Rechten. Liefert im Erfolgsfall eine 1 zurück andernfalls eine 0 und setzt `$!(errno)`.

```
mkdir("sweets",0755) || die "cannot mkdir
sweets: $!";
```

Removing Directories

`rmdir(DIRNAME)`

`rmdir DIRNAME`

Löscht das Verzeichnis `DIRNAME`, falls es leer ist. Liefert 1, wenn es geklappt hat, sonst 0 setzt dann `$! errno`).

```
rmdir("sweets") || die "cannot rmdir sweets:$!";
```

Rechte in UNIX/Linux

ls -l

```
jupiter:/home/Debra > ls -l
total 92
drwxr-xr-x    2 bernd    users          35 2003-10-25 11:43 Dokumente
drwxr-xr-x    2 bernd    users          35 2003-10-25 11:44 Mail
drwxr-xr-x    7 bernd    users         287 2003-10-25 13:18 Kursunterlagen
-rw-r--r--    1 bernd    users         187 2003-11-10 19:36 zitat.txt
-rw-r--r--    1 bernd    users       46992 2003-11-11 19:09 shell.jpg
-rwxr--r--    1 bernd    users       42288 2003-11-11 19:09 pipe.gif

jupiter:/home/Debra >
```

1	2	3	4	5	6	7	8	9	10
Type	User Permissions			Group Permissions			Other Permissions		
Type	Read	Write	Execute	Read	Write	Execute	Read	Write	Execute
d	r	w	x	r	-	x	r	-	x
	4	2	1						

Rechte ändern

In Linux/UNIX werden die Rechte einer Datei oder eines Verzeichnisses mit dem Kommando `chmod` geändert.

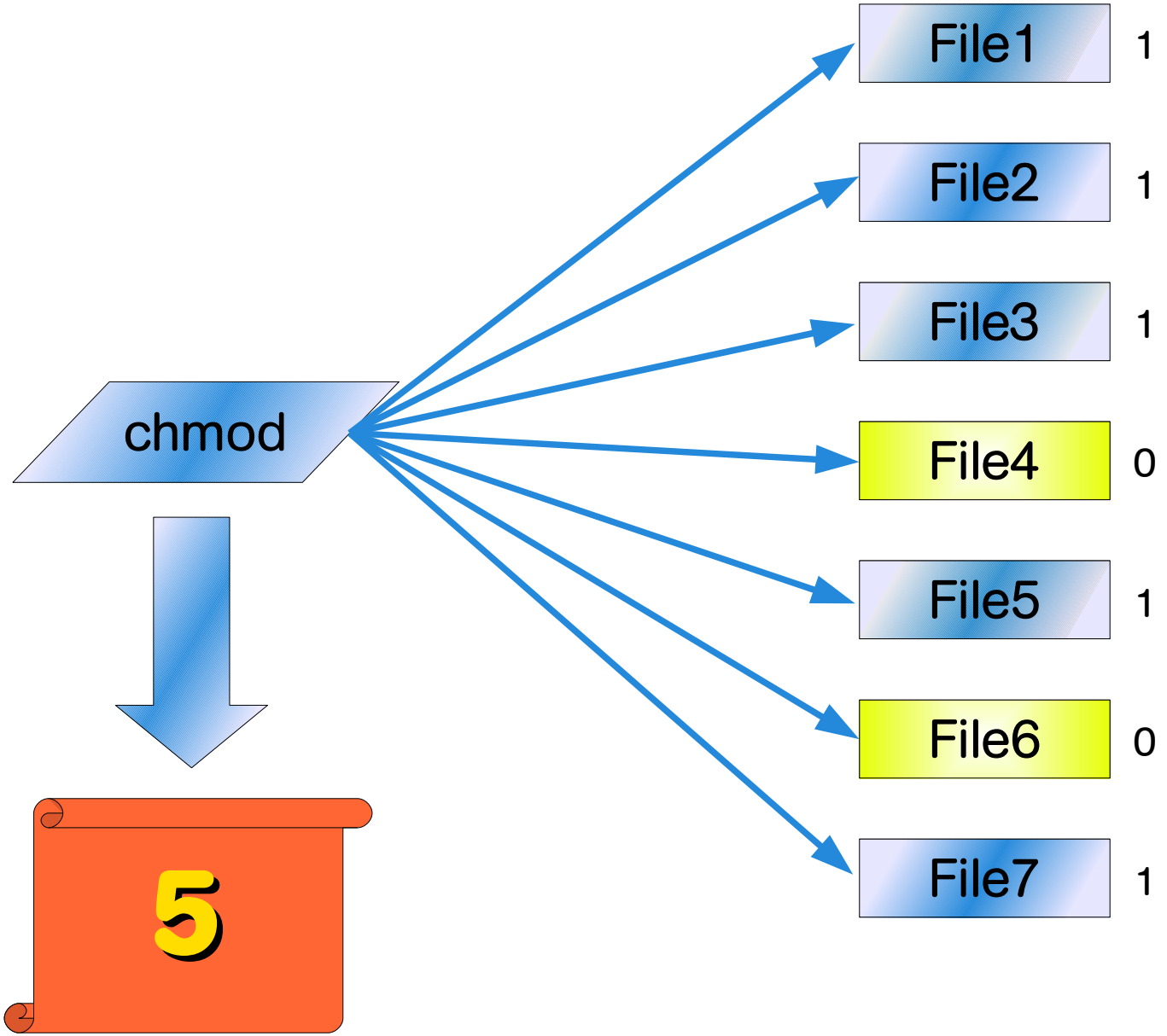
Auch Perl benutzt dazu eine Funktion gleichen Namens.

Diese Rechte bestimmen, wer was mit einer Datei oder einem Verzeichnis tun darf.

Die chmod-Function benötigt einen oktalen numerischen Wert als den “Modus (mode)” und eine Liste von Datei- und Verzeichnisnamen. “Perl” versucht die Rechte dieser Datei- und Verzeichnisnamen entsprechend zu ändern.

```
chmod(0644,”homer”,”lisa”,”bart”,”marge”);
```

chmod



Rückgabewert von chmod

Der Rückgabewert von chmod ist die Anzahl der erfolgreich angepassten Dateien- und Verzeichnisse. Dabei spielt es keine Rolle ob es dadruch wirklich zu einer Änderung kommt.

```
foreach $file ("homer","marge") {
    unless chmod (0644,$file) {
        warn "Couldn't chmod $file: $!";
    }
}
```

Wechselnde Eigentümer-Verhältnisse

chown

Jede Datei, Verzeichnis, Gerät oder was auch hat einen Besitzer (owner) und gehört zu einer Gruppe.

Der Besitzer und die Gruppe einer Datei oder eines Verzeichnisses werden bei der Erzeugung bereits vergeben

Häufig möchte man diese jedoch später wieder ändern.

Dazu gibt es in Perl den folgenden Befehl:

```
chown LIST
```

Die Liste `LIST` besteht aus UID, GID und einer Liste von zu ändernden Datei- und/oder Verzeichnisnamen:

```
chown(1004, 4711, “apples”, “oranges”,”bananas”);
```