

Perl für Computerlinguisten

Matthias Bethke

`bethke@linguistik.uni-erlangen.de`

Linguistische Informatik
Universität Erlangen-Nürnberg

Sommersemester 2007

- 1 Organisatorisches
- 2 Einführung
- 3 Das erste Programm
- 4 Perl-Dokumentation

Organisatorisches

- 1 Termin
- 2 Voraussetzungen und Ablauf
- 3 Übungsaufgaben
- 4 Literatur
- 5 WWW: <http://www.linguistik.uni-erlangen.de/~msbethke/teaching/SS2007-PerlCL.html>

Beliebte Vorurteile

Perl ist...

- Unlesbar
- Nur was für Hacker
- Langsam
- Veraltet

In Wahrheit...

- Erlaubt unlesbaren Code, aber auch strukturierten, objektorientierten, ...
- → *TIMTOWTDI* („There is more than one way to do it“)
- Für eine Skriptsprache relativ schnell
- Sehr gut geeignet für Textverarbeitung (Systemadministration, NLP, Web-Anwendungen, etc.)
- In den letzten 20 Jahren kontinuierlich weiterentwickelt
- Riesige Bibliothek von Programmmodulen verfügbar (→CPAN, „Comprehensive Perl Archive Network“)

Allgemeines

- Skriptsprache: kompiliert unmittelbar vor der Laufzeit
→ Kurze Entwicklungszyklen („mal eben...“)
→ Laufzeitnachteil gegenüber kompilierten Sprachen
- Entwickelt als mächtigere Alternative zu den Unix-Tools `sed`, `awk` und `sh` für die Systemadministration
- In der Computerlinguistik beliebt v.a. wegen umfangreicher Möglichkeiten zur Textmanipulation und regulärer Ausdrücke
- Einfache Schnittstellen zu anderen Programmen (z.B. Malaga)

Allgemeines zu Perl-Skripten

- Einfache Textdateien, üblicherweise mit der Endung „.pl“ (nicht notwendig)
- Vorsicht mit verschiedenen Zeilenumbrüchen (DOS vs. Unix), kann zu rätselhaften Fehlern führen

Allgemeines zu Perl-Skripten

- Einfache Textdateien, üblicherweise mit der Endung „.pl“ (nicht notwendig)
- Vorsicht mit verschiedenen Zeilenumbrüchen (DOS vs. Unix), kann zu rätselhaften Fehlern führen

Kopfzeile zur Angabe des Interpreters

```
#!/usr/bin/perl [Optionen] # -w ist immer gut!  
# Hier kommt das eigentliche Skript
```

Anschließend: `chmod +x meinskript.pl`.

Allgemeines zu Perl-Skripten

- Einfache Textdateien, üblicherweise mit der Endung „.pl“ (nicht notwendig)
- Vorsicht mit verschiedenen Zeilenumbrüchen (DOS vs. Unix), kann zu rätselhaften Fehlern führen

Kopfzeile zur Angabe des Interpreters

```
#!/usr/bin/perl [Optionen] # -w ist immer gut!  
# Hier kommt das eigentliche Skript
```

Anschließend: `chmod +x meinskript.pl`.

Alternativen auf der Kommandozeile

```
perl [Optionen] meinskript.pl  
perl [Optionen] -e '<Anweisungen>'
```

Hello, world!

Das üblicherweise erste Programm

```
#!/usr/bin/perl -w
my $hello = "Hello, world!";
print $hello . "\n";
```

Hello, world!

Das üblicherweise erste Programm

```
#!/usr/bin/perl -w
my $hello = "Hello, world!";
print $hello . "\n";
```

- Befehle werden mit Semikolon abgeschlossen.

Hello, world!

Das üblicherweise erste Programm

```
#!/usr/bin/perl -w
my $hello = "Hello, world!";
print $hello . "\n";
```

- Befehle werden mit Semikolon abgeschlossen.
- *Skalarvariablen* beginnen mit dem Dollarzeichen (`$foo`)

Hello, world!

Das üblicherweise erste Programm

```
#!/usr/bin/perl -w
my $hello = "Hello, world!";
print $hello . "\n";
```

- Befehle werden mit Semikolon abgeschlossen.
- *Skalarvariablen* beginnen mit dem Dollarzeichen (`$foo`)
- Variablen *müssen nicht* deklariert werden, *sollten* aber! („`use strict;`“ vor dem Programmcode erzwingt Deklaration)
 - Einzel: `my $variable;`

Hello, world!

Das üblicherweise erste Programm

```
#!/usr/bin/perl -w
my $hello = "Hello, world!";
print $hello . "\n";
```

- Befehle werden mit Semikolon abgeschlossen.
- *Skalarvariablen* beginnen mit dem Dollarzeichen (`$foo`)
- Variablen *müssen nicht* deklariert werden, *sollten* aber! („`use strict;`“ vor dem Programmcode erzwingt Deklaration)
 - Einzel: `my $variable;`
 - Mehrere: `my ($var1, $var2, $var3);`

Hello, world!

Das üblicherweise erste Programm

```
#!/usr/bin/perl -w
my $hello = "Hello, world!";
print $hello . "\n";
```

- Befehle werden mit Semikolon abgeschlossen.
- *Skalarvariablen* beginnen mit dem Dollarzeichen (`$foo`)
- Variablen *müssen nicht* deklariert werden, *sollten* aber! („`use strict;`“ vor dem Programmcode erzwingt Deklaration)
 - Einzel: `my $variable;`
 - Mehrere: `my ($var1, $var2, $var3);`
 - Initialisierung (=Wertzuweisung) schon bei der Deklaration möglich

Hello, world!

Das üblicherweise erste Programm

```
#!/usr/bin/perl -w
my $hello = "Hello, world!";
print $hello . "\n";
```

- Befehle werden mit Semikolon abgeschlossen.
- *Skalarvariablen* beginnen mit dem Dollarzeichen (`$foo`)
- Variablen *müssen nicht* deklariert werden, *sollten* aber! („use strict;“ vor dem Programmcode erzwingt Deklaration)
 - Einzel: `my $variable;`
 - Mehrere: `my ($var1, $var2, $var3);`
 - Initialisierung (=Wertzuweisung) schon bei der Deklaration möglich
- Erster *Operator*: Stringkonkatenation (`$hello . "\n"`)

Hello, world!

Das üblicherweise erste Programm

```
#!/usr/bin/perl -w
my $hello = "Hello, world!";
print $hello . "\n";
```

- Befehle werden mit Semikolon abgeschlossen.
- *Skalarvariablen* beginnen mit dem Dollarzeichen (\$foo)
- Variablen *müssen nicht* deklariert werden, *sollten* aber! („use strict;“ vor dem Programmcode erzwingt Deklaration)
 - Einzel: my \$variable;
 - Mehrere: my (\$var1, \$var2, \$var3);
 - Initialisierung (=Wertzuweisung) schon bei der Deklaration möglich
- Erster *Operator*: Stringkonkatenation (\$hello . "\n")
- Eingebaute funktion print zur Textausgabe.

Dokumentation

Je nach System findet sich die sehr umfangreiche Perl-Dokumentation an verschiedenen Stellen:

- 1 Man-Pages: `man perl` gibt eine Übersicht über die verfügbaren Handbucheinträge. Für den Anfang am wichtigsten:
 - `perlintro` Einführung für Anfänger
 - `perlsyn` Syntaxreferenz
 - `perlfunc` Referenz zu den eingebauten Funktionen
- 2 Mittels `perldoc` (`man perl` bzw. `man perltoc` für ein Inhaltsverzeichnis)
- 3 Unter Windows als HTML-Dateien im Installationsverzeichnis

Variablen in Perl (1)

Perl ist eine *schwach typisierte Sprache* und kommt mit 3 bzw. 5 Datentypen aus:

- 1 Skalare: Präfix \$
 - Speichern Strings und Zahlenwerte (Referenzen: später)
 - Keine Unterscheidung zwischen Integer- und Fließkommazahlen, nicht einmal zwischen Zahlen und Strings (legal: `print "3.5"* 3`)

Variablen in Perl (1)

Perl ist eine *schwach typisierte Sprache* und kommt mit 3 bzw. 5 Datentypen aus:

- 1 Skalare: Präfix \$
 - Speichern Strings und Zahlenwerte (Referenzen: später)
 - Keine Unterscheidung zwischen Integer- und Fließkommazahlen, nicht einmal zwischen Zahlen und Strings (legal: `print "3.5"* 3`)
- 2 Listen (Arrays): Präfix @, z.B. „@mylist“
 - Eindimensionale Sammlungen von skalaren Werten
 - Einzelne Werte werden über Integer-Indizes in eckigen Klammern angesprochen, z.B. `print $array[5]`

Variablen in Perl (2)

- 3 Hashes (assoziative Arrays): Präfix %, z.B. „%hash_map“ (näheres später)

Variablen in Perl (2)

- ③ Hashes (assoziative Arrays): Präfix %, z.B. „%hash_map“ (näheres später)
- ④ Subroutinen (Funktionen): Präfix &, z.B. „&tolleFunktion“ (eigentlich kein Datentyp, wird aber im selben Zusammenhang gebraucht)
- ⑤ Typeglob: Präfix *, z.B. „*foo“ (Fortgeschrittenenthema)

Variablen in Perl (2)

- ③ Hashes (assoziative Arrays): Präfix %, z.B. „%hash_map“ (näheres später)
- ④ Subroutinen (Funktionen): Präfix &, z.B. „&tolleFunktion“ (eigentlich kein Datentyp, wird aber im selben Zusammenhang gebraucht)
- ⑤ Typeglob: Präfix *, z.B. „*foo“ (Fortgeschrittenenthema)
- Variablen gleichen Namens und unterschiedlichen Typs sind möglich, wenn auch nicht empfehlenswert: `my ($a, @a);`

Variablen in Perl (2)

- ③ Hashes (assoziative Arrays): Präfix %, z.B. „%hash_map“ (näheres später)
- ④ Subroutinen (Funktionen): Präfix &, z.B. „&tolleFunktion“ (eigentlich kein Datentyp, wird aber im selben Zusammenhang gebraucht)
- ⑤ Typeglob: Präfix *, z.B. „*foo“ (Fortgeschrittenenthema)
- Variablen gleichen Namens und unterschiedlichen Typs sind möglich, wenn auch nicht empfehlenswert: `my ($a, @a);`
- Generelle Regel beim Ansprechen einzelner Elemente: entscheidend für den benutzten Präfix ist der Ziel-Typ!
`@array1 = @array2;` → Ganzes Array kopieren
`$foo = $array2[1];` → Einzelnes Element (Skalar!) kopieren

Strings und *Interpolation*

Die Benutzung verschiedener Anführungszeichen entscheidet in Perl über die Interpretation des Inhalts.

- Einfache Anführungszeichen („Hochkommas“): Strings *ohne Interpolation*, d.h. Zeichen werden interpretiert genau wie sie dastehen: `$s = 'Das_kostet_$100';`

- Doppelte („normale“) Anführungszeichen: Strings *mit Interpolation*, d.h. Variablen werden eingefügt und *Escape-Sequenzen* interpretiert:

```
$preis = 'hundert';  
$text = "Das_kostet_$preis_Euro\n";
```

- *Backticks*: Inhalt wird als Name eines Programms interpretiert und die Ausgabe dieses Programms eingefügt:
`$vz = `ls -l`;`

Übung (1)

- Geben Sie das Helloworld-Programm ein und probieren Sie es aus.
- Legen Sie eine weitere Skalarvariable an und lassen Sie sie in den Ausgabestring interpolieren.
- Lassen sie diese Variable vor dem `print`-Befehl mit der Ausgabe des Kommandos „`uname -a`“ befüllen.

Übung (2)

- Berechnen Sie den Durchschnitt von drei Zahlenwerten (Tip: die mathematischen Operatoren $+$, $-$, $*$ und $/$ funktionieren wie gewohnt; es gilt „Punkt vor Strich“) und geben Sie ihn aus. Falls das Ergebnis nicht stimmt: lieber einmal zu viel klammern als zu wenig!
- Wandeln Sie eine Anzahl Sekunden in eine Ausgabe in Stunden:Minuten:Sekunden um.
Tip: der Modulo-Operator „%“ ergibt den Rest einer Division.

Eingabe vom Terminal

- Um Daten von außerhalb des Programms verarbeiten zu können, müssen diese aus Dateien gelesen werden.
- Die Dateien `STDIN`, `STDOUT` und `STDERR` beziehen sich auf das Terminal, in dem das Programm läuft und müssen nicht extra geöffnet werden.

Eingabe vom Terminal

- Um Daten von außerhalb des Programms verarbeiten zu können, müssen diese aus Dateien gelesen werden.
- Die Dateien `STDIN`, `STDOUT` und `STDERR` beziehen sich auf das Terminal, in dem das Programm läuft und müssen nicht extra geöffnet werden.
- Der sog. *Diamond-Operator* „`<>`“ liest zeilenweise aus einer Datei:
`$variable = <STDIN>;`
→ `$variable` enthält eine Zeile Text, sobald der Benutzer diese eingegeben und die Eingabetaste gedrückt hat.

Eingabe vom Terminal

- Um Daten von außerhalb des Programms verarbeiten zu können, müssen diese aus Dateien gelesen werden.
- Die Dateien `STDIN`, `STDOUT` und `STDERR` beziehen sich auf das Terminal, in dem das Programm läuft und müssen nicht extra geöffnet werden.
- Der sog. *Diamond-Operator* „`<>`“ liest zeilenweise aus einer Datei:
`$variable = <STDIN>;`
→ `$variable` enthält eine Zeile Text, sobald der Benutzer diese eingegeben und die Eingabetaste gedrückt hat.
- Vorsicht: das Zeilenvorschubzeichen gehört mit zur Eingabe! Wenn es (wie üblich) unerwünscht ist, kann es mit der eingebauten Funktion `chomp` entfernt werden: **`chomp $variable;`**

Übung

Schreiben Sie ein Programm, das eine Anzahl Meilen einliest und sie in Kilometer umrechnet. Der Umrechnungsfaktor beträgt 1.609.

Listen

- Listenindizes zählen von null ab, also `$liste[0]`, `$liste[1]`, ...
- Deklaration und Initialisierung einer Liste:
`my @zeug = ("Laptop", "Zettel", "Stift");`
- Einzige Bedingung für Listenelemente: sie müssen Skalare sein:
`my @mischmasch = (0, "Liste", "", 5.3e5, "Ende");`

Listen

- Listenindizes zählen von null ab, also `$liste[0]`, `$liste[1]`, ...
- Deklaration und Initialisierung einer Liste:
`my @zeug = ("Laptop", "Zettel", "Stift");`
- Einzige Bedingung für Listenelemente: sie müssen Skalare sein:
`my @mischmasch = (0, "Liste", "", 5.3e5, "Ende");`
- Zuweisen einer Liste an mehrere Skalare, umständlich und einfach:
`my $null = $zeug[0];`
`my $eins = $zeug[1];`
`my $zwei = $zeug[2];`

Listen

- Listenindizes zählen von null ab, also `$liste[0]`, `$liste[1]`, ...
- Deklaration und Initialisierung einer Liste:
`my @zeug = ("Laptop", "Zettel", "Stift");`
- Einzige Bedingung für Listenelemente: sie müssen Skalare sein:
`my @mischmasch = (0, "Liste", "", 5.3e5, "Ende");`
- Zuweisen einer Liste an mehrere Skalare, umständlich und einfach:
`my $null = $zeug[0];`
`my $eins = $zeug[1];`
`my $zwei = $zeug[2];`
`my ($null, $eins, $zwei) = @zeug;`

Listen (2)

- Listenmanipulation

- Am Listenende: ←push / pop →

push @list, \$elem;

\$elem = **pop** @list;

- Am Listenanfang: ←shift / unshift →

\$elem = **shift** @list;

unshift @list, \$elem;

Listen (2)

- Listenmanipulation

- Am Listenende: \leftarrow push / pop \rightarrow

push @list, \$elem;

\$elem = **pop** @list;

- Am Listenanfang: \leftarrow shift / unshift \rightarrow

\$elem = **shift** @list;

unshift @list, \$elem;

- Teilstück („slice“) aus einer Liste kopieren:

@slice = @list[3 .. 5];

\rightarrow „..“: allgemeiner Bereichsoperator, erzeugt eine Liste von Integerzahlen start .. ende. **Achtung**: Leerzeichen links und rechts!

\rightarrow Teilstück ist eine Liste, deshalb hier mit „@“

Listen (3)

- Teillisten (so) nicht möglich:
(1, 2, (3, 4), 5) == (1, 2, 3, 4, 5)
Grund: Listen werden ineinander „interpoliert“.

Listen (3)

- Teillisten (so) nicht möglich:
(1, 2, (3, 4), 5) == (1, 2, 3, 4, 5)
Grund: Listen werden ineinander „interpoliert“.
- Bequeme Schreibweise für einfache Listenelemente: **qw()**
(„quote words“):
@tiere = **qw**(Hund Katze Maus Ameisenbaer);

Kontext

Perl-Anweisungen und Variablen verhalten sich oft unterschiedlich, je nachdem in welchem Kontext sie eingesetzt werden, d.h. welcher Datentyp von einem Operator, einer Funktion oder einer Zuweisung verlangt wird. Man unterscheidet:

1 Skalkontext

- Z.B.: `$var = @list;` oder **print** `@list . "n";`
- Unterteilt in String-, numerischen und „egal“-Kontext (meist irrelevant)
- Kann erzwungen werden: **print scalar**(`@list`);

2 Listenkontext

- Z.B. `print Ausdruck;` (`print` verlangt eine Liste);
(`$bla`, `Ausdruck`, `$fael`); (Interpolation von `Ausdruck`)
`@lines = 'ls';` (Zuweisung an eine Liste)
- Dass eine Anweisung ihre Argumente im Listenkontext auswertet, ist im Handbuch mit LIST vermerkt.

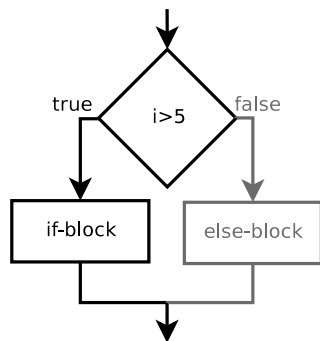
Übung

- Verschaffen Sie sich im Perl-Manual einen Überblick über die verfügbaren Kapitel und Tutorials.
- Schlagen Sie in der Funktionsreferenz die `print`-Anweisung und ihre verschiedenen Aufrufmöglichkeiten nach
- Schlagen Sie die `join`- und `reverse`-Anweisungen nach und probieren Sie sie aus (`reverse`: Listen und Strings!)
- Definieren Sie eine unsortierte Liste von Zahlen und versuchen Sie sie mittels „`sort @liste`“ zu sortieren. Was ist das Problem, und was sagt das Handbuch dazu?

Kontrollstrukturen, die erste: if-else

Zum Steuern des Programmflusses in Abhängigkeit von Daten brauchen wir Kontrollstrukturen. Die einfachste: if-else.

```
if ($i > 5) {  
  # Befehle, die ausgeführt werden  
  # wenn $i > 5 ist  
} else {  
  # Befehle, die ausgeführt werden  
  # wenn $i <= 5 ist  
}
```



Bedingungen (1)

- Bedingungen sind *boolesche Ausdrücke*, d.h. Ausdrücke, die zu *wahr* oder *falsch* ausgewertet werden können.
 - Eine uninitialisierte Variable hat den Wert **undef**, der immer *falsch* ergibt.
 - Leere Strings und die 0 ergeben *falsch*.
 - Nichtleere Strings und von 0 verschiedene Zahlen ergeben *wahr*.

Bedingungen (1)

- Bedingungen sind *boolesche Ausdrücke*, d.h. Ausdrücke, die zu *wahr* oder *falsch* ausgewertet werden können.
 - Eine uninitialisierte Variable hat den Wert **undef**, der immer *falsch* ergibt.
 - Leere Strings und die 0 ergeben *falsch*.
 - Nichtleere Strings und von 0 verschiedene Zahlen ergeben *wahr*.
- Vergleichsoperatoren: ==/>/</>=/<= (numerisch gleich/größer/kleiner/größer oder gleich/kleiner oder gleich)

Bedingungen (1)

- Bedingungen sind *boolesche Ausdrücke*, d.h. Ausdrücke, die zu *wahr* oder *falsch* ausgewertet werden können.
 - Eine uninitialisierte Variable hat den Wert **undef**, der immer *falsch* ergibt.
 - Leere Strings und die 0 ergeben *falsch*.
 - Nichtleere Strings und von 0 verschiedene Zahlen ergeben *wahr*.
- Vergleichsoperatoren: `==/ > / < / >= / <=` (numerisch gleich/größer/kleiner/größer oder gleich/kleiner oder gleich)
- Negationsoperator: `!`
Zum Beispiel: `if(! $i > 5) { ... }`

Bedingungen (1)

- Bedingungen sind *boolesche Ausdrücke*, d.h. Ausdrücke, die zu *wahr* oder *falsch* ausgewertet werden können.
 - Eine uninitialisierte Variable hat den Wert **undef**, der immer *falsch* ergibt.
 - Leere Strings und die 0 ergeben *falsch*.
 - Nichtleere Strings und von 0 verschiedene Zahlen ergeben *wahr*.
- Vergleichsoperatoren: `==/ >/ </ >= / <=` (numerisch gleich/größer/kleiner/größer oder gleich/kleiner oder gleich)
- Negationsoperator: `!`
Zum Beispiel: `if(! $i > 5) { ... }`
- **unless** ist „syntaktischer Zucker“ für `if(! ...)`.

Bedingungen (2)

Weitere Operatoren

- String-Vergleich: `eq`, `ne`, `lt` und `gt`
z.B: `$lessThan = "Hund"lt "Katze";`

Bedingungen (2)

Weitere Operatoren

- String-Vergleich: `eq`, `ne`, `lt` und `gt`
z.B: `$lessThan = "Hund"lt "Katze";`
- Vergleichsoperatoren zum sortieren: `<=>` (numerisch) und `cmp` (textuell)

Bedingungen (2)

Weitere Operatoren

- String-Vergleich: `eq`, `ne`, `lt` und `gt`
z.B: `$lessThan = "Hund"lt "Katze";`
- Vergleichsoperatoren zum sortieren: `<=>` (numerisch) und `cmp` (textuell)
- Logische Operatoren ! (Negation, s. o.), `&&/and`: UND, `||/or`: ODER, `xor`: Exklusiv-ODER.
 - Achtung: nicht verwechseln mit Bit-Operatoren „&“ bzw. „|“!

Bedingungen (2)

Weitere Operatoren

- String-Vergleich: `eq`, `ne`, `lt` und `gt`
z.B: `$lessThan = "Hund"lt "Katze";`
- Vergleichsoperatoren zum sortieren: `<=>` (numerisch) und `cmp` (textuell)
- Logische Operatoren ! (Negation, s. o.), `&&/and`: UND, `||/or`: ODER, `xor`: Exklusiv-ODER.
 - Achtung: nicht verwechseln mit Bit-Operatoren „&“ bzw. „|“!
 - Unterschied zwischen `&&/and` u. ä.: Operatorvorrang, ähnlich „Punkt-vor-Strich“: `and/or` „binden“ schwächer, d. h. sie werden in einem komplexen Ausdruck später ausgeführt.

Übung

- Erweitern Sie das Meilenumrechnungsprogramm so, dass es zuerst eine Maximalzahl von Kilometern einliest und bei deren Überschreitung „Zu weit!“ ausgibt.
- Stellen Sie fest, was passiert, wenn der Diamond-Operator nichts zu lesen hat (Eingabeumleitung aus `/dev/null` benutzen) und berücksichtigen Sie diesen Fall bei der Verarbeitung der Eingabe.

Mehr Kontrollstrukturen: while und do-while

```
while($i < 3) {
  # Befehle, die ausgefuehrt werden
  # solange $i < 3 ist
}
```

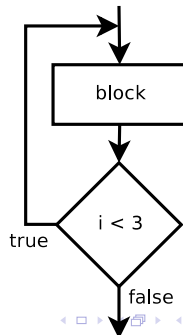
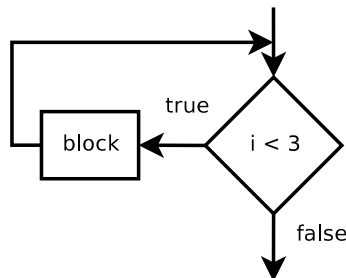
Negierte Alternativform:

```
until( Bedingung ) { Anweisungen }
```

```
do {
  # Befehle, die ausgefuehrt werden
  # solange $i < 3 ist
} while($i < 3);
```

Negierte Alternativform:

```
do { Anweisungen } until( Bedingung );
```



Logikoperatoren: „und“, „oder“ und „exklusiv-oder“

Wahrheitstabellen logischer Operatoren:

a	b	a AND b	a	b	a OR b	a	b	a XOR b
F	F	F	F	F	F	F	F	F
F	W	F	F	W	W	F	W	W
W	F	F	W	F	W	W	F	W
W	W	W	W	W	W	W	W	F

Logikoperatoren: „und“, „oder“ und „exklusiv-oder“

Wahrheitstabellen logischer Operatoren:

a	b	a AND b	a	b	a OR b	a	b	a XOR b
F	F	F	F	F	F	F	F	F
F	W	F	F	W	W	F	W	W
W	F	F	W	F	W	W	F	W
W	W	W	W	W	W	W	W	F

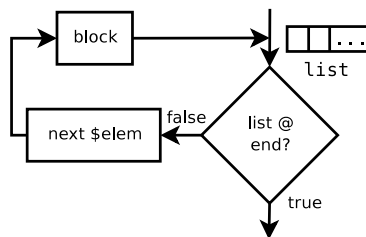
- Perl-Operatoren `&&` (and), `||` (or), `^` (xor)
- Verknüpfung boolescher Ausdrücke:
`if($a > 5 || ($b == 0 && $s eq "foo"))` → „wenn entweder \$a größer als 5 *oder* sowohl \$b gleich 0 als auch \$s gleich 'foo' ist, dann...“
- Zur Flusskontrolle: `$a == 1 and print "a_ist_1\n";` oder `$b > 2 or print "b_ist_kleiner/gleich_2\n";`
 Warum funktioniert das? (vgl. Wahrheitstabellen!)
 Warum ist das mit „xor“ sinnlos?

Übung

- Schreiben Sie ein Programm, das bis 10 zählt und dabei jedes mal den Zähler ausgibt.
- Wie kann man mit `while` oder `do-while` eine Endlosschleife konstruieren? Die Anweisung „`last`“ bricht eine Schleife ab – lassen Sie sich eine Abbruchbedingung einfallen und verlassen Sie so eine Endlosschleife „per Hand“.
- Mit Hilfe von Schleifen und *Diamond*-Operator können wir bereits Textdateien zeilenweise bearbeiten. Wie muss dafür die Abbruchbedingung aussehen? Schreiben Sie ein Programm, das eine Textdatei zeilenweise einliest, dabei die Zeilen zählt und sie nummeriert ausgibt.

Listeniteration mit foreach

```
foreach $scalar (@liste) {
  # Schleifenkoerper
}
```



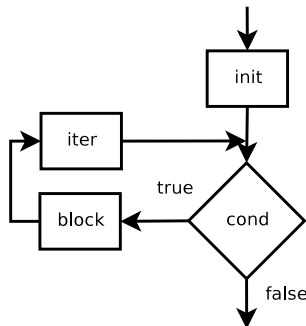
- Anweisungen werden für jedes Element in @liste wiederholt, wobei \$scalar auf das jeweilige Element verweist.
- **Achtung:** \$scalar ist *keine Kopie* sondern ein sog. *Alias* des jeweiligen Listenelements! D. h. jede Modifikation desselben verändert @liste selbst.

Zählschleifen mit for

Abstrakt:

```
for( init ; cond; iter ) {
  # Schleifenkoerper
}
```

```
for(my $i=10; $i>5; --$i) {
  print "$i\n";
}
```



- Die Zählschleife entspricht weitestgehend der aus C/C++/Java bekannten Form. **for** und **foreach** sind in Perl Synonyme, die nur der Verständlichkeit halber unterschieden werden.
- „init“, „cond“ und „iter“ können beinahe beliebige Anweisungen enthalten oder auch ganz entfallen. Die Semikolons sind obligatorisch.

Kurzformen der Kontrollstrukturen

- Im Gegensatz zu C/Java sind die geschweiften Klammern obligatorisch!

```
# Falsch
if ($a == 1)
    print "a_ist_eins\n";
# Richtig
if ($a == 1) {
    print "a_ist_eins\n";
}
```

Kurzformen der Kontrollstrukturen

- Im Gegensatz zu C/Java sind die geschweiften Klammern obligatorisch!

```
# Falsch
if($a == 1)
  print "a_ist_eins\n";
# Richtig
if($a == 1) {
  print "a_ist_eins\n";
}
```

- Aber (Ausnahme bei Einzelanweisungen):

```
print "a_ist_eins\n"if($a == 1);
exit(1) unless($a);
```

- Genauso:

```
print "Element: $_\n"foreach(@liste);
```

Die letzteren Konstruktionen dienen der „natürlichsprachlicheren“ Formulierung von Code.

Achtung: bei **foreach** nur mit *Default-Variable* (später...)

Kontrollstrukturen verlassen: `next` und `last`

Zum vorzeitigen Verlassen eines Schleifenablaufs gibt es zwei Möglichkeiten:

- „`next`“ bricht den *Schleifenkörper* ab und beginnt den nächsten Durchlauf mit der *Schleifenbedingung*.
- „`last`“ bricht die Schleife komplett ab und setzt das Programm mit dem nächsten Befehl fort.

Um geschachtelte Schleifen zu verlassen bzw. neu zu starten, werden *Labels* benötigt:

Labels

```
OUTER: for(my $i=0; $i<10; ++$i) {  
    for(my $j=42; $j>0; --$j) {  
        # ...  
        last OUTER if($error);  
    }  
}
```

Dateien (1)

Dateien werden über *Dateihandles* angesprochen, die i.d.R. explizit geöffnet werden müssen. Dateihandles haben *keinen Typpräfix* (\$) o. ä.) und werden konventionell GROSS geschrieben.

- Automatisch geöffnete Handles: STDIN, STDOUT und STDERR

Dateien (1)

Dateien werden über *Dateihandles* angesprochen, die i.d.R. explizit geöffnet werden müssen. Dateihandles haben *keinen Typpräfix* (\$) o. ä.) und werden konventionell GROSS geschrieben.

- Automatisch geöffnete Handles: STDIN, STDOUT und STDERR
- Öffnen zum lesen: **open**(HANDLE,"name") gibt **undef** für Fehler zurück.

open(): konservativ, perliger und idiomatisch

```
if (!open(FH,"dateiname")) {  
    die "dateiname:␣$!";  
}
```

Dateien (1)

Dateien werden über *Dateihandles* angesprochen, die i.d.R. explizit geöffnet werden müssen. Dateihandles haben *keinen Typpräfix* (\$) o. ä.) und werden konventionell GROSS geschrieben.

- Automatisch geöffnete Handles: STDIN, STDOUT und STDERR
- Öffnen zum lesen: **open**(HANDLE,"name") gibt **undef** für Fehler zurück.

open(): konservativ, perliger und idiomatisch

```
if (!open(FH,"dateiname")) {  
    die "dateiname:␣$!";  
}  
  
die "dateiname:␣$!" unless(open(FH,"dateiname"));
```

Dateien (1)

Dateien werden über *Dateihandles* angesprochen, die i.d.R. explizit geöffnet werden müssen. Dateihandles haben *keinen Typpräfix* (\$) o. ä.) und werden konventionell GROSS geschrieben.

- Automatisch geöffnete Handles: STDIN, STDOUT und STDERR
- Öffnen zum lesen: **open**(HANDLE,"name") gibt **undef** für Fehler zurück.

open(): konservativ, perliger und idiomatisch

```
if (!open(FH,"dateiname")) {  
    die "dateiname:␣$!";  
}  
  
die "dateiname:␣$!" unless(open(FH,"dateiname"));  
  
open(FH,"dateiname") or die "dateiname:␣$!";
```

Dateien (1)

Dateien werden über *Dateihandles* angesprochen, die i.d.R. explizit geöffnet werden müssen. Dateihandles haben *keinen Typpräfix* (\$) o. ä.) und werden konventionell GROSS geschrieben.

- Automatisch geöffnete Handles: STDIN, STDOUT und STDERR
- Öffnen zum lesen: **open**(HANDLE,"name") gibt **undef** für Fehler zurück.

open(): konservativ, perliger und idiomatisch

```
if (!open(FH,"dateiname")) {
    die "dateiname:␣$!";
}

die "dateiname:␣$!" unless(open(FH,"dateiname"));

open(FH,"dateiname") or die "dateiname:␣$!";
```

- Schließen: automatisch bei Programmende, oder (besser) mit **close**(HANDLE);

Dateien (2)

Der Modus zum Öffnen einer Datei kann als Teil des Dateinamens übergeben werden (→Shell-Syntax zur Ein-/Ausgabeumleitung!)

- Lesen: **open**(LIES, "dateiname");
oder: **open**(LIES, "<dateiname");
- Schreiben: **open**(SCHREIB, "dateiname");
Falls die Datei nicht existiert, wird sie neu angelegt.
- Anhängen: **open**(ANHAENG, ">>dateiname");
Anlegen wie oben

Dateien (2)

Der Modus zum Öffnen einer Datei kann als Teil des Dateinamens übergeben werden (→Shell-Syntax zur Ein-/Ausgabeumleitung!)

- Lesen: **open**(LIES, "dateiname");
oder: **open**(LIES, "<dateiname");
- Schreiben: **open**(SCHREIB, "dateiname");
Falls die Datei nicht existiert, wird sie neu angelegt.
- Anhängen: **open**(ANHAENG, ">>dateiname");
Anlegen wie oben
- Pipe an ein Kommando: **open**(PIPERAUS, "|befehl");
„befehl“ wird ausgeführt und erhält in PIPERAUS geschriebene Daten auf der Standardeingabe.
- Pipe von einem Kommando: **open**(PIPEREIN, "befehl|");
Von „befehl“ ausgegebene Daten erscheinen zum lesen in PIPEREIN.

Dateien (3)

Schreiben in Dateien

- STDOUT ist Default
- Sonst mit „`print HANDLE LIST`“ (s. Funktionsreferenz!)

Lesen aus Dateien üblicherweise zeilen- bzw. datensatzweise mit dem *Diamond-Operator* „`<>`“

- `<>` liest von `STDIN`¹, `<HANDLE>` vom angegebenen Dateihandle.

¹Vorsicht mit `@ARGV`, s. `man perl`, Abschnitt „I/O Operators“! Besser explizit `<STDIN>` verwenden.

Dateien (3)

Schreiben in Dateien

- STDOUT ist Default
- Sonst mit „`print HANDLE LIST`“ (s. Funktionsreferenz!)

Lesen aus Dateien üblicherweise zeilen- bzw. datensatzweise mit dem *Diamond-Operator* „`<>`“

- `<>` liest von `STDIN`¹, `<HANDLE>` vom angegebenen Dateihandle.
- Der Operator ist kontextsensitiv
 - Im Skalkontext wird bei jedem Aufruf eine Zeile (bzw. ein Datensatz) zurückgeliefert, am Dateiende undef.
→ `while($zeile = <HANDLE>) ...`
 - Im Listekontext wird die komplette Datei auf einmal gelesen und in einzelne Sätze/Zeilen zerlegt. → `@datei = <HANDLE>;`

¹Vorsicht mit `@ARGV`, s. man `perl op`, Abschnitt „I/O Operators“! Besser explizit `<STDIN>` verwenden.

Übung

- Das Array @ARGV enthält die dem Skript übergebenen Kommandozeilenargumente. Gehen Sie dieses Array elementweise durch und geben Sie die Argumente aus.
- Interpretieren Sie jedes Argument als einen Dateinamen und versuchen Sie die Datei zu öffnen. Falls eine Datei nicht existiert, soll eine Fehlermeldung auf STDERR ausgegeben und ggf. mit der nächsten forgefahren werden.
- Lesen Sie die Dateien jeweils in ein Array ein und geben Sie sie aus.
- Geben Sie die Zeilen rückwärts (d.h. die letzte Zeile zuerst) aus. Ist dafür das einlesen im Skalar- oder Listenkontext geeigneter?

Die Defaultvariable \$_

- Viele Anweisungen akzeptieren eine reduzierte Zahl von Argumenten und arbeiten dann üblicherweise auf der *Defaultvariablen*.

Die Defaultvariable \$_

- Viele Anweisungen akzeptieren eine reduzierte Zahl von Argumenten und arbeiten dann üblicherweise auf der *Defaultvariablen*.
- Die Verwendung ist eine Stilfrage: als kurz und knapp beliebt, als kryptisch und reine Faulheit verschrien.

Die Defaultvariable \$_

- Viele Anweisungen akzeptieren eine reduzierte Zahl von Argumenten und arbeiten dann üblicherweise auf der *Defaultvariablen*.
- Die Verwendung ist eine Stilfrage: als kurz und knapp beliebt, als kryptisch und reine Faulheit verschrien.

Mit Defaultvariablen	Mit expliziten Variablen
<pre>while(<>) { chomp; s/abc/def/; print foreach(split); }</pre>	<pre>while(my \$line = <>) { chomp \$line; \$line =~ s/abc/def/; foreach my \$s (split / /,\$line) { print \$s; } }</pre>

Funktionen: Definition

- Eine Funktion fasst mehrere Anweisungen zu einem unter einem definierten Namen ansprechbaren und wiederverwendbaren Block zusammen.

Funktionen: Definition

- Eine Funktion fasst mehrere Anweisungen zu einem unter einem definierten Namen ansprechbaren und wiederverwendbaren Block zusammen.
- Funktionen können beliebig viele *Argumente* (auch: *Parameter*) erhalten und ein Ergebnis liefern.
- Keine Unterscheidung zwischen Funktionen und Prozeduren.

Funktionen: Definition

- Eine Funktion fasst mehrere Anweisungen zu einem unter einem definierten Namen ansprechbaren und wiederverwendbaren Block zusammen.
- Funktionen können beliebig viele *Argumente* (auch: *Parameter*) erhalten und ein Ergebnis liefern.
- Keine Unterscheidung zwischen Funktionen und Prozeduren.
- Allg. Definition: `sub name { <Anweisungen> }`
- „Prototypen“: s. *Programmieren mit Perl* S. 233 (engl. S. 118).
Zum Beispiel `funktion($$)`; für zwei Skalar-Argumente oder `funktion($@)`; für zwei Skalare und eine Liste. Prototypen müssen vor der ersten Verwendung der Funktion im Programm erscheinen!

Funktionen: Argumente

Argumente werden nicht benannt sondern jeder Funktion im implizit deklarierten Array @_ übergeben. Dieses kann auf verschiedene Arten verwendet werden:

- 1 Argumentarray direkt verwenden:

```
# Argumente zurueckgeben, die groesser als 10 sind
```

```
sub args1 {  
  return grep {$_ > 10} @_;  
}
```

Funktionen: Argumente

Argumente werden nicht benannt sondern jeder Funktion im implizit deklarierten Array `@_` übergeben. Dieses kann auf verschiedene Arten verwendet werden:

- 1 Argumentarray direkt verwenden:

```
# Argumente zurueckgeben, die groesser als 10 sind
```

```
sub args1 {
  return grep {$_ > 10} @_;
}
```

- 2 shiften von `@_`:

```
sub args2 {
  my $arg = shift; # implizites "shift @_!"
  # ...
}
```

Funktionen: Argumente

Argumente werden nicht benannt sondern jeder Funktion im implizit deklarierten Array @_ übergeben. Dieses kann auf verschiedene Arten verwendet werden:

- 1 Argumentarray direkt verwenden:

```
# Argumente zurueckgeben, die groesser als 10 sind
sub args1 {
  return grep {$_ > 10} @_;
}
```

- 2 shiften von @_:

```
sub args2 {
  my $arg = shift; # implizites "shift @_"!
  # ...
}
```

- 3 Array-Zuweisung, am besten bei mehreren Argumenten:

```
sub args3 {
  my ($arg1, $arg2, $arg3) = @_;
  # ...
}
```

Funktionen: Verwendung

- Funktionsaufruf:
 - `subname(LIST)` geht immer.
 - `subname LIST` geht, falls die Funktion vorher deklariert wurde.
 - `&subname` : explizit als Subroutine aufrufen.

Funktionen: Verwendung

- Funktionsaufruf:
 - `subname(LIST)` geht immer.
 - `subname LIST` geht, falls die Funktion vorher deklariert wurde.
 - `&subname` : explizit als Subroutine aufrufen.
- Werte werden *by-reference* übergeben, d.h. Änderungen am Argument-Array wirken sich direkt auf die Argumente aus, sofern diese veränderlich sind.

Ausprobieren:

```
my $x = 0;  
sub byref { $_[0] = 5; }  
byref($x);  
print "$x\n";
```

Funktionen: Verwendung

- Funktionsaufruf:
 - `subname(LIST)` geht immer.
 - `subname LIST` geht, falls die Funktion vorher deklariert wurde.
 - `&subname` : explizit als Subroutine aufrufen.
- Werte werden *by-reference* übergeben, d.h. Änderungen am Argument-Array wirken sich direkt auf die Argumente aus, sofern diese veränderlich sind.

Ausprobieren:

```
my $x = 0;
sub byref { $_[0] = 5; }
byref($x);
print "$x\n";
```

- Rückgabe von Werten aus einer Funktion:


```
return <Wert>;
```
- Zurückgegeben werden können Skalare (**return** `$a_value`;) oder Listen (**return** `@a_list`;).

Lokale Variable

- Mit „**my**“ deklarierte Variable in einem *Block* sind *lokal*, d.h. nur innerhalb des Blocks gültig.
- Verlässt der Programmfluss den Block, „verschwinden“ diese lokalen Variablen, sind also nicht mehr verwendbar, und ihr Speicher wird freigegeben.
- Ausnahme: solange es noch Referenzen (später!) auf eine Variable gibt, verschwindet nur der lokale Name, die Variable bleibt aber erhalten und über die Referenz ansprechbar.

```
sub localvars {  
    my $s = "test\n";  
    print $s;    # Gibt "test" aus  
}  
localvars ;  
print $s;      # Gibt nichts aus (Warnung wegen $s==undef)
```

Übung

- Schreiben Sie das Programm zum rückwärts-ausgeben von Dateien so um, dass die Operationen zum öffnen und lesen der Datei in eine Funktion ausgelagert sind.
- Funktionen können sich auch in Perl selbst aufrufen, also *rekursiv* sein. Schreiben Sie eine Funktion, die die Fibonacci-Zahlen bis zu einem übergebenen Maximalwert ausgibt ($\text{fib}(0)=0$; $\text{fib}(1)=1$; $\text{fib}(n)=\text{fib}(n-2)+\text{fib}(n-1)$).

Hashes

Der dritte und letzte noch fehlende Datentyp, auch „assoziatives Array“ genannt.

- Im Unterschied zum Array speichern Hashes Werte nicht unter *numerischen Indizes* sondern *beliebigen Schlüsseln*.
→Der Inhalt eines Hashes sind immer *Attribut-Werte-Paare!*

Hashes

Der dritte und letzte noch fehlende Datentyp, auch „assoziatives Array“ genannt.

- Im Unterschied zum Array speichern Hashes Werte nicht unter *numerischen Indizes* sondern *beliebigen Schlüsseln*.
→Der Inhalt eines Hashes sind immer *Attribut-Werte-Paare!*
- Syntax:
 - Typpräfix: Prozentzeichen (%), z.B. `my %hash;`
 - Anlegen: `my %kurse = ("Besim" => "EMSV", "Matthias" => "Perl");`
 - Zugriff: `$meinKurs = $kurse{"Matthias"};`

Hashes

Der dritte und letzte noch fehlende Datentyp, auch „assoziatives Array“ genannt.

- Im Unterschied zum Array speichern Hashes Werte nicht unter *numerischen Indizes* sondern *beliebigen Schlüsseln*.
→Der Inhalt eines Hashes sind immer *Attribut-Werte-Paare!*
- Syntax:
 - Typpräfix: Prozentzeichen (%), z.B. `my %hash;`
 - Anlegen: `my %kurse = ("Besim" => "EMSV", "Matthias" => "Perl");`
 - Zugriff: `$meinKurs = $kurse{"Matthias"};`
 - Der Pfeiloperator `,=>` ist nur ein Synonym für das Komma, es wird dem Hash also eine normale Liste (runde Klammern!) zugewiesen.
 - Unterschied zum Komma: die linke Seite ist automatisch ein String, Anführungszeichen können also entfallen: `(Matthias => "Perl")`
 - Ebenso in geschweiften Klammern: `$kurse{Matthias}`

Übung

Das erste *brauchbare* Perl-Skript für die CL!

- Gegeben ist ein „vertikalisierte“ Text (d.h. ein Wort pro Zeile) unter `/korpora/Limas/limas.pp`.
- Überlegen Sie sich, wie ein Hash zum Zählen von Wortformfrequenzen dienen kann.
- Implementieren Sie diese Methode für das o.a. Limas-Korpus.

Reguläre Ausdrücke (1): Allgemeines

- Reguläre Ausdrücke sind eine der großen Stärken von Perl und werden in fast jedem Programm verwendet.
- Die Syntax entspricht weitgehend der von `egrep(1)` („V8 Regular Expressions“), mit einigen Erweiterungen
- Ausführliche Beschreibung mit `man perlre`
- Regexp-Operatoren wirken bezüglich der Interpolation wie Strings in Anführungszeichen, d.h. Variablen werden vor der Auswertung interpoliert.
- Nicht explizit an eine Variable gebundene Ausdrücke arbeiten wie üblich auf der Defaultvariablen.
- Bei den folgenden Operatoren können statt Schrägstrichen auch andere Zeichen (`#`, `!` etc.) benutzt werden. Vorsicht: keine Interpolation bei Hochkommata (`'`)!

Reguläre Ausdrücke (2): Match / Bindung

- `m//`: „Match“-Operator, testet auf Übereinstimmung mit einem RA.
 - `m/fo{2,}bar/`;
 - `m!~/usr/bin/!`; (weitere Begrenzerzeichen z. B. #, =, \$)
 - Werden Schrägstriche verwendet, kann das ‚m‘ wegfallen:

```
while(<>) {  
    print unless /^#/;  
}
```

Reguläre Ausdrücke (2): Match / Bindung

- `m//`: „Match“-Operator, testet auf Übereinstimmung mit einem RA.

- `m/fo{2,}bar/`;
- `m!~/usr/bin/!`; (weitere Begrenzerzeichen z. B. #, =, \$)
- Werden Schrägstriche verwendet, kann das ‚m‘ wegfallen:

```
while(<>) {
  print unless /^#/;
}
```

- `=~`: Bindungsoperator, bindet eine Variable zur Bearbeitung an einen RA. Zum Beispiel:

```
$s =~ m#[a-z]([0-9])#;
```

Reguläre Ausdrücke (3): Substitution

`s///`: „Substitute“-Operator, ersetzt Vorkommen eines regulären Ausdrucks durch String oder Ausdruck

Reguläre Ausdrücke (3): Substitution

`s///`: „Substitute“-Operator, ersetzt Vorkommen eines regulären Ausdrucks durch String oder Ausdruck

- Links (zwischen linkem und mittlerem Begrenzer): regulärer Ausdruck; rechts: Ersatz für jedes Vorkommen dieses Ausdrucks.
- Flags (hinter rechtem Begrenzer) geben Voreinstellungen zum Abpassen des Ausdrucks an.
- Wichtige Flags (weitere siehe Manpage):
 - `i`: *case-insensitive* (Groß-/Kleinschreibung ignorieren)
 - `g`: „globales“ Ersetzen, d.h. alle Vorkommen des RA statt nur das erste
 - `o`: Ausdruck nur einmal compilieren (→ Laufzeitvorteil, Interpolation!)
 - `e`: rechte Seite als Ausdruck interpretieren

Reguläre Ausdrücke (3.1): Substitution: Beispiele

- `s/[abc].*x/def/`
Erstes Vorkommen von „`[abc].*x`“ durch „`def`“ ersetzen.

Reguläre Ausdrücke (3.1): Substitution: Beispiele

- `s/[abc] .*x/def/`
Erstes Vorkommen von „`[abc] .*x`“ durch „`def`“ ersetzen.
- `s!foo!bar!ogi`
Alle Vorkommen von „`foo`“ durch „`bar`“ ersetzen –
Groß-/Kleinschreibung ignorieren und RA nur einmal
compilieren.

Reguläre Ausdrücke (3.1): Substitution: Beispiele

- `s/[abc] .*x/def/`
Erstes Vorkommen von „`[abc] .*x`“ durch „`def`“ ersetzen.
- `s!foo!bar!ogi`
Alle Vorkommen von „`foo`“ durch „`bar`“ ersetzen – Groß-/Kleinschreibung ignorieren und RA nur einmal compilieren.
- `$s =~ s#([a-z])([0-9])#$2$1#g`
`$1/$2`: Platzhalter für Klammerausdrücke

Reguläre Ausdrücke (3.1): Substitution: Beispiele

- `s/[abc] .*x/def/`
Erstes Vorkommen von „`[abc] .*x`“ durch „`def`“ ersetzen.
- `s!foo!bar!ogi`
Alle Vorkommen von „`foo`“ durch „`bar`“ ersetzen – Groß-/Kleinschreibung ignorieren und RA nur einmal compilieren.
- `$s =~ s#([a-z])([0-9])#$2$1#g`
`$1/$2`: Platzhalter für Klammerausdrücke
- `$s =~ s/blah/&fasel1/`
Erstes Vorkommen von „`blah`“ durch „`blahfasel`“ ersetzen.

Reguläre Ausdrücke (3.1): Substitution: Beispiele

- `s/[abc] .*x/def/`
Erstes Vorkommen von „[abc] .*x“ durch „def“ ersetzen.
- `s!foo!bar!ogi`
Alle Vorkommen von „foo“ durch „bar“ ersetzen – Groß-/Kleinschreibung ignorieren und RA nur einmal compilieren.
- `$s =~ s#([a-z])([0-9])#$2$1#g`
\$1/\$2: Platzhalter für Klammerausdrücke
- `$s =~ s/blah/&fasel/`
Erstes Vorkommen von „blah“ durch „blahfasel“ ersetzen.
- `$s =~ s!\d+!$&*2!eg`
Matcht Sequenzen von Ziffern und verdoppelt die resultierende Zahl (Flag "e" → Ausdruck rechts!)

Reguläre Ausdrücke (3.2): Idioms

- Der Match-Operator gibt im Listenkontext eine Liste der Teilstrings zurück, die auf Klammerausdrücke gepasst haben:

Match-Idiom

```
my $s = "Das ist das Haus vom Nikolaus";  
($erstes, $letztes) = $s =~ /^(\w+).* (\w+)$/;
```

- Erstes und letztes Wort werden den entsprechenden Variablen zugewiesen.

Reguläre Ausdrücke (3.2): Idioms

- Der Match-Operator gibt im Listenkontext eine Liste der Teilstrings zurück, die auf Klammerausdrücke gepasst haben:

Match-Idiom

```
my $s = "Das_ist_das_Haus_vom_Nikolaus";
($erstes, $letztes) = $s =~ /^(\w+).* (\w+)/;
```

- Erstes und letztes Wort werden den entsprechenden Variablen zugewiesen.

Leere-Klammern-Idiom

```
my $s = "Das_ist_das_Haus_vom_Nikolaus";
$anz = () = $s =~ /\S+/g;
```

- [g] globales Matchen auf Sequenzen von Nicht-Whitespace *im Listenkontext*, Zuweisung trotzdem an einen Skalar!

Reguläre Ausdrücke (3.2): Idioms

- Der Match-Operator gibt im Listenkontext eine Liste der Teilstrings zurück, die auf Klammerausdrücke gepasst haben:

Match-Idiom

```
my $s = "Das_ist_das_Haus_vom_Nikolaus";
($erstes, $letztes) = $s =~ /^(\w+).* (\w+)/;
```

- Erstes und letztes Wort werden den entsprechenden Variablen zugewiesen.

Leere-Klammern-Idiom

```
my $s = "Das_ist_das_Haus_vom_Nikolaus";
$anz = () = $s =~ /\S+/g;
```

- [g] globales Matchen auf Sequenzen von Nicht-Whitespace *im Listenkontext*, Zuweisung trotzdem an einen Skalar!

Übung

Öffnen Sie die Datei `/korpora/Limas/limas.byb.latin-1` bzw. `/korpora/Limas/limas.byb.utf-8` (je nach Ihrer Zeichensatzeinstellung) und verarbeiten Sie sie.

- Entfernen Sie alle Zeilennummernmarkierungen am Zeilenanfang.
- Ersetzen Sie Umlaute durch entsprechende Ersatzdarstellungen (`ä`→`ae`, `ß`→`ss`, etc.)
- Geben Sie die Daten mit einem Satz pro Zeile aus (Tip: Zeilen konkatenieren und jedes Mal prüfen, ob ein mit Leerzeichen abgetrennter Punkt vorkommt!)

Referenzen: das Problem

Erweitern des bisherigen Frequenzlistengenerators um Frequenzklassen.

- Gewünschte Ausgabe: eine Zeile pro Frequenzklasse
- Dahinter die jeweiligen Wortformen, die entsprechend oft im Korpus vorkommen.
- Möglichst nicht eine Zeile für $1 \dots \text{MAX_FREQUENZ}$ sondern nur die tatsächlich vorkommenden.

Referenzen: das Problem

Erweitern des bisherigen Frequenzlistengenerators um Frequenzklassen.

- Gewünschte Ausgabe: eine Zeile pro Frequenzklasse
- Dahinter die jeweiligen Wortformen, die entsprechend oft im Korpus vorkommen.
- Möglichst nicht eine Zeile für $1 \dots \text{MAX_FREQUENZ}$ sondern nur die tatsächlich vorkommenden.
- Mögliche Lösung: Umsortieren des Hash in eine Liste von Listen.
- Aber: Listen sind an sich immer eindimensional!

Referenzen (1)

- Perl-Äquivalent zu Zeigern in C².
- *Skalare* „Verweise“ auf andere Daten beliebigen Typs.
- Grundlage für beliebig komplexe Datenstrukturen und Objekte in der OOP.

²In Java sind sowieso *alle* Objektdatentypen in Wirklichkeit Referenzen

Referenzen (1)

- Perl-Äquivalent zu Zeigern in C².
- *Skalare* „Verweise“ auf andere Daten beliebigen Typs.
- Grundlage für beliebig komplexe Datenstrukturen und Objekte in der OOP.
 - Allg. Syntax: Backslash-Operator erzeugt eine Referenz, z. B.:
`$scalarref = \ $scalar;`
`$arrayref = \@array;`
`$hashref = \%hash;`

²In Java sind sowieso *alle* Objektdatentypen in Wirklichkeit Referenzen

Referenzen (1)

- Perl-Äquivalent zu Zeigern in C².
- *Skalare* „Verweise“ auf andere Daten beliebigen Typs.
- Grundlage für beliebig komplexe Datenstrukturen und Objekte in der OOP.

- Allg. Syntax: Backslash-Operator erzeugt eine Referenz, z. B.:

```
$scalarref = \$scalar;
```

```
$arrayref = \@array;
```

```
$hashref = \%hash;
```

- *Anonyme Variablen*:

```
Arrays:           $anona = ["foo", "bar"];
```

```
Mit Interpolation: $anona = [ qw(foo bar) ];
```

```
Hashes:          $anonh = {Key1 => "Va11", Key2 => "Va12"};
```

²In Java sind sowieso *alle* Objektdatentypen in Wirklichkeit Referenzen

Referenzen (2)

- Dereferenzieren durch Voranstellen des Typpräfixes des referenzierten Typs (*-Operator in C):

```
$realscalar = $$scalarref;
```

```
@realarray = @$arrayref;
```

```
%realhash = %$hashref;
```

- Zugriff auf Elemente einer referenzierten Struktur:

```
$arrayelem = $arrayref->[n];
```

```
$hashelem = $hashref->{foo};
```

Referenzen (2)

- Dereferenzieren durch Voranstellen des Typpräfixes des referenzierten Typs (*-Operator in C):
`$realscalar = $$scalarref;`
`@realarray = @$arrayref;`
`%realhash = %$hashref;`
- Zugriff auf Elemente einer referenzierten Struktur:
`$arrayelem = $arrayref->[n];`
`$hashelem = $hashref->{foo};`
- Disambiguierung durch geschweifte Klammern:
`$${$var[0]}`: erstes Element von `@var` als Skalarreferenz betrachten und dereferenzieren.
- Merke: ein Block in `{...}` hat den Wert, der zuletzt darin angegeben wurde.

Zwischenspiel

Zunächst überlegen, dann ausprobieren:

```
my $s = "test";  
my $sref = \$s;  
print "$s\n$sref\n$$sref"; # Ausgabe?
```

```
my $aref = [ "foo", "bar" ];  
print "$_\n" foreach( ... ); # Ergaenzen
```

```
my %h = ( key => 42 );  
my $href = \%h;  
my $hrefref = \$href;  
# Unter "key" gespeicherten Wert per $hrefref ausgeben
```

Übung

Gegeben ist die folgende Definition einer Datenstruktur. Stellen Sie die Struktur grafisch dar und geben Sie in Perl die Elemente 'sur' und die komplette Wortstruktur aus.

Komplexe Datenstruktur

```
my $analysis = {  
  sur => 'bauern',  
  base => 'bauer',  
  structure => [  
    { Morpheme => 'bauen', Allomorph => 'bau' POS: 'Verb' },  
    { Morpheme => 'er', Allomorph => 'er', POS => 'Suffix' },  
    { Morpheme => 'n', Allomorph => 'n', POS => 'Suffix' }  
  ]  
  pos => 'noun'  
}
```

Anwendung auf das Frequenzklassenproblem

Benötigt:

- Datenstruktur zum Zugriff auf einzelne Frequenzklassen
- Liste von Einträgen für jede einzelne Klasse

Anwendung auf das Frequenzklassenproblem

Benötigt:

- Datenstruktur zum Zugriff auf einzelne Frequenzklassen
- Liste von Einträgen für jede einzelne Klasse

Lösung: Listenreferenzen in anderer Datenstruktur.

Anwendung auf das Frequenzklassenproblem

Benötigt:

- Datenstruktur zum Zugriff auf einzelne Frequenzklassen
- Liste von Einträgen für jede einzelne Klasse

Lösung: Listenreferenzen in anderer Datenstruktur.

Vorüberlegung: Array oder Hash für die Arrayreferenzen?

- Pro Array: schnellerer Zugriff, Index ist ohnehin numerisch
- Pro Hash: geringerer Speicherverbrauch bei ungewöhnlichen Frequenzverteilungen

Alte und neue Struktur

- Bisherige Frequenzliste:

```
aal      5
aalborg 1
aale     3
aalen   1
```

- Neue Frequenzklassenliste:

```
10:[ abdomen, abdruck, abendessen, ...]
9: [ abrechen, abfassung, abgebrochen, ...]
8: [ abbauen, abeler, aberglauben, ...]
```

Übung

- Schreiben Sie das Frequenzlistenprogramm auf Frequenzklassenausgabe um
- Probieren Sie ein Hash und ein Array als primäre Datenstruktur aus. Worauf ist bei Arrays besonders zu achten?

Code-Referenzen

Referenzen können auch auf Funktionen verweisen. Hiermit sind echte *Closures* wie in funktionalen Programmiersprachen möglich.

Einfache Codereferenz

```
my $coderef = sub { print "Args:␣@_\n"; };
&$coderef("hallo");
&$coderef(qw(1 2 3 foo));
```

Echte Closure

```
sub makeAdder {
  my $n = shift;
  return sub { return $_[0] + $n; }
}
$add1 = makeAdder(1);
$add42 = makeAdder(42);
print &$add1(2), "\n";
print &$add42(10), "\n";
```

Module: allgemeines

- F: Wie macht man *xyz* in Perl?
A: Dafür gibt's ein Modul auf CPAN³.

³Comprehensive Perl Archive Network, <http://www.cpan.org/>

⁴Eigentlich *Pragmas*! Aus Benutzerperspektive egal. 

Module: allgemeines

- F: Wie macht man *xyz* in Perl?
A: Dafür gibt's ein Modul auf CPAN³.
- Module verpacken Funktionalität in wiederverwendbarer Form
- Mittlerweile meistens in objektorientiertem Perl geschrieben
- Einbinden eines Moduls mittels „**use**“;

³Comprehensive Perl Archive Network, <http://www.cpan.org/>

⁴Eigentlich *Pragmas*! Aus Benutzerperspektive egal. 

Module: allgemeines

- F: Wie macht man *xyz* in Perl?
A: Dafür gibt's ein Modul auf CPAN³.
- Module verpacken Funktionalität in wiederverwendbarer Form
- Mittlerweile meistens in objektorientiertem Perl geschrieben
- Einbinden eines Moduls mittels „**use** ;“
- Wichtige mitgelieferte Module⁴:
 - **use strict**; aktiviert strengere Überprüfungen z.B. bezüglich Variablen- und Funktionsdeklaration.
 - **use warnings**; warnt beim Aufruf undefinierter Funktionen, Verwendung undefinierter Werte etc. (äquivalent zum Interpreteraufruf mit „-w“)
 - **use diagnostics**; aktiviert ausführlichere Fehlermeldungen.
 - **use English**; erlaubt sprechendere Namen für Spezialvariablen wie \$!, \$/ etc.

³Comprehensive Perl Archive Network, <http://www.cpan.org/>

⁴Eigentlich *Pragmas*! Aus Benutzerperspektive egal 

Module benutzen

- Hierarchische Struktur, auf Dateisystemebene in Verzeichnissen, in Perl mit Doppelpunkten getrennt, z.B. Modul installiert als `Lingua/Ident.pm` → Einbinden mit **use** `Lingua::Ident`;

Module benutzen

- Hierarchische Struktur, auf Dateisystemebene in Verzeichnissen, in Perl mit Doppelpunkten getrennt, z.B. Modul installiert als `Lingua/Ident.pm` → Einbinden mit `use Lingua::Ident;`
- Kleine Auswahl wichtiger Module für die Computerlinguistik
 - `locale`, `encoding`, `utf8`: Internationalisierungs- und UTF8-Funktionen.
 - `Lingua::*`, Module zur Sprachverarbeitung, z.B. `Lingua::Stem::De`: deutscher Stemming-Algorithmus
 - `HTML::*` und `XML::*` alles zur Verarbeitung von HTML- bzw. XML-Dateien, z.B. `XML::Writer`: Ausgabemodul für XML-Dateien.
 - `Encode::*`: Codierungsfunktionen; Umgang mit verschiedenen Zeichensätzen
 - CLUE: `Ma1aga`: LA-Parser.

Objektorientierte Programmierung

- Objekte sind aus Benutzerperspektive Referenzen
- Klasse einbinden durch Benutzen des entsprechenden Moduls, z.B. `use IO::File;`
- Konstruktion: `my $obj = KLASSE->new();`, z.B.:
`my $outfile = IO::File->new(">NeueDatei");`
oder (TIMTOWTDI!)
`my $outfile = new IO::File(">NeueDatei");`

Objektorientierte Programmierung

- Objekte sind aus Benutzerperspektive Referenzen
- Klasse einbinden durch Benutzen des entsprechenden Moduls, z.B. `use IO::File;`
- Konstruktion: `my $obj = KLASSE->new();`, z.B.:
`my $outfile = IO::File->new(">NeueDatei");`
oder (TIMTOWTDI!)
`my $outfile = new IO::File(">NeueDatei");`
- Methodenaufruf ebenfalls über die Referenzsyntax:
`$outfile->read($buffer,1000);`

Beispiel: XML-Ausgabe einer Frequenzliste

- `man XML::Writer!`

Beispiel: XML-Ausgabe einer Frequenzliste

- `man XML::Writer!`
- Ausgangspunkt: Frequenzklassen-Programm der letzten Stunde
- Vorgehen:
 - 1 XML-Format überlegen!

Beispiel: XML-Ausgabe einer Frequenzliste

- `man XML::Writer!`
- Ausgangspunkt: Frequenzklassen-Programm der letzten Stunde
- Vorgehen:
 - 1 XML-Format überlegen!
 - 2 Ausgabedatei mittels `IO::File` öffnen.
 - 3 Neues `XML::Writer`-Objekt erzeugen (Konfiguration mittels übergebenem Hash; `DATA_MODE?`)
 - 4 XML-Deklaration für Zeichensatz anlegen
 - 5 Start- und Endtag für Dokument erzeugen

Beispiel: XML-Ausgabe einer Frequenzliste

- man `XML::Writer!`
- Ausgangspunkt: Frequenzklassen-Programm der letzten Stunde
- Vorgehen:
 - 1 XML-Format überlegen!
 - 2 Ausgabedatei mittels `IO::File` öffnen.
 - 3 Neues `XML::Writer`-Objekt erzeugen (Konfiguration mittels übergebenem Hash; `DATA_MODE?`)
 - 4 XML-Deklaration für Zeichensatz anlegen
 - 5 Start- und Endtag für Dokument erzeugen
 - 6 Für jede Frequenzklasse Start- und End-Tag erzeugen; dazwischen einzelne Wortformen ausgeben (Methoden `startTag`, `endTag`, `dateElement`).
- Fürs Tutorium: komplett objektorientierte Version des Frequenzklassengenerators entwerfen und so weit wie möglich programmieren (z. B. `FClass.pm` mit Konstruktor, `add()`- und `toXML`-Methoden, etc.)

Eigene Module und Klassen schreiben

- Am besten in einzelne Dateien aufteilen (wie Java); Module mit Endung `.pm`.
- Module müssen nicht unbedingt objektorientiert arbeiten, auch wenn dies mittlerweile üblich ist.
- `package`-Deklaration dem Dateinamen entsprechend → eigener Namensraum
- Keine vorgegebenen Methodennamen, auch Konstruktoren können beliebig benannt werden.
- Unterklassen mit `use base Oberklassenliste` (Mehrfachvererbung!);
- Klassen- bzw. statische Methoden bekommen den Klassennamen als erstes Argument, Objektmethoden die Objektreferenz.
- Ein Modul muss `true` zurückgeben, um erfolgreich importiert zu werden, üblicherweise mit „1;“ in der letzten Zeile.

Eine Beispielklasse

MyClass.pm

```

package MyClass;
sub new {
    my $class = shift;           # Klassenname als Argument
    my $self = bless {}, $class; # "Dieses Hash ist ein Objekt der
                                # Klasse $class"!
    $self->{counter} = 0;        # Ein Attribut
    return $self;               # Objektreferenz zurueckgeben
}
sub count {
    my ($self, $increment) = @_;
    $self->{counter} += $increment || 42;    # Attribut aendern
}
sub reset {
    shift->{counter} = 0;        # Auch "shift" hat einen Wert
}
1;

```

Die Beispielklasse benutzen

MyClass.pm

```
use MyClass;
my $cnt = MyClass->new;  # Objekt erzeugen
print $cnt->count(5), "\n"; # Mal mit Argument...
print $cnt->count, "\n";  # ...mal ohne
$cnt->reset;
print $cnt->count, "\n";
```

Unterklassen

MySubclass.pm

```

package MySubclass;
use base 'MyClass'; # oft: qw(MyClass) -> Mehrfachvererbung!
sub new {
    my $self = shift -> SUPER::new(@_); # Konstruktor der Oberklasse
    $self -> {resets} = 0;                # Neues Attribut
    return $self;
}
sub reset {                                # reset-Methode ueberladen
    my $self = shift;
    $self -> SUPER::reset; # Die Originalfunktion aufrufen
    ++$self -> {resets};  # Zusaezliche Funktionalitaet
}
1;

```