Perl-Programmierung für Linguisten

Matthias Bethke
bethke@linguistik.uni-erlangen.de

Linguistische Informatik
Universität Erlangen-Nürnberg

Sommersemester 2007

Organisatorisches

1. Termin und Ablauf
2. Spezielle Interessen der Teilnehmer
3. Übungsaufgaben
4. Literatur
5. WWW: `http://www.linguistik.uni-erlangen.de/~msbethke/teaching/SS2007-PerlLing.html`

Literatur

- Joachim Ziegler: *Programmieren lernen mit Perl*. Springer 2002 (ISBN 354042685X)
  Führt am Beispiel von Perl in die Programmierung allgemein ein. Solide Grundlagenkapitel und gute Übungsaufgaben.
- Eidenberger, H. und Michlmayr, E.: *Mit Perl programmieren lernen*. dpunkt.verlag 2005 (ISBN 3898643204)
  Pragmatisches Einsteigerbuch mit kleinen Schwächen aber guter Einführung für Nicht-Informatiker.
- Wall, L., Christansen, T. und Orwant, J.: *Programmieren mit Perl*. O'Reilly 2001 (ISBN 3897211440)
  Die Referenz von den Erfindern von Perl, als Einsteigerbuch recht anspruchsvoll aber immer wieder nützlich.

Programmieren: warum überhaupt?

- Üblicherweise: „Mein Programm kann X nicht!“

Programmieren: warum überhaupt?

- Üblicherweise: „Mein Programm kann X nicht!“
- Oder: es gibt überhaupt kein Programm, das die gewünschte Tätigkeit erledigt.

Programmieren: warum überhaupt?

- Üblicherweise: „Mein Programm kann X nicht!“
- Oder: es gibt überhaupt kein Programm, das die gewünschte Tätigkeit erledigt.
- Aber auch: zur geistig-moralischen Erbauung :)

Programmieren: warum überhaupt?

- Üblicherweise: „Mein Programm kann X nicht!“
- Oder: es gibt überhaupt kein Programm, das die gewünschte Tätigkeit erledigt.
- Aber auch: zur geistig-moralischen Erbauung :)

Und:

- Computer sind zwar sehr schnell, aber auch sehr schwer von Begriff.

$\longrightarrow$ Man muss ihnen alles, was nicht unmittelbar logisch herzuleiten ist, haarklein vorschreiben.

Algorithmen

- Genau definierte Anweisungen zur Lösung eines Problems in endlich vielen Schritten
- Benannt nach der Rechenanleitung des Persers Muhammed Al Chwarizmi, um 783

[1] Idealisierte Abstraktion eines Computers

Algorithmen

- Genau definierte Anweisungen zur Lösung eines Problems in endlich vielen Schritten
- Benannt nach der Rechenanleitung des Persers Muhammed Al Chwarizmi, um 783
- i. A. zur Ausführung auf einem Computer gedacht, kann sich aber auch auf Kochrezepte, Bedienungsanleitungen u. dgl. beziehen.

[1] Idealisierte Abstraktion eines Computers

Algorithmen

- Genau definierte Anweisungen zur Lösung eines Problems in endlich vielen Schritten
- Benannt nach der Rechenanleitung des Persers Muhammed Al Chwarizmi, um 783
- i. A. zur Ausführung auf einem Computer gedacht, kann sich aber auch auf Kochrezepte, Bedienungsanleitungen u. dgl. beziehen.
- Formulierung je nach Zweck natürlichsprachlich, abstrakt als Flussdiagramm o. ä., konkret als Programmcode in einer bestimmten Programmiersprache, oder konkret/mathematisch als Maschinenprogramm einer Turingmaschine[1]

[1] Idealisierte Abstraktion eines Computers

Beispiel-Algorithmus (1)[2]

Natürlichsprachlich

„Suche in einer Zeitschrift nach Wohnungsinseraten. Enthält eine Zeitung ausreichend viele interessante Inserate, soll sie abonniert werden.“

[2] Nach Eidenberger 2005

Beispiel-Algorithmus (1)[2]

Natürlichsprachlich

„Suche in einer Zeitschrift nach Wohnungsinseraten. Enthält eine Zeitung ausreichend viele interessante Inserate, soll sie abonniert werden.“

Aufteilen des Problems in Schritte

1. Zeitschrift zur Hand nehmen
2. Zeitschrift lesen
3. Inhalt der Artikel bewerten
4. Nützlichkeit der Zeitschrift bewerten

[2] Nach Eidenberger 2005

Beispiel-Algorithmus (1)[2]

Natürlichsprachlich

„Suche in einer Zeitschrift nach Wohnungsinseraten. Enthält eine Zeitung ausreichend viele interessante Inserate, soll sie abonniert werden."

Aufteilen des Problems in Schritte

1. Zeitschrift zur Hand nehmen
2. Zeitschrift lesen
3. Inhalt der Artikel bewerten
4. Nützlichkeit der Zeitschrift bewerten

- Wie funktioniert das suchen?
- Was macht einen interessanten Artikel aus?

[2] Nach Eidenberger 2005

Beispiel-Algorithmus (2)

Erste Präzisierung

1. Zeitschrift aufschlagen
2. Wiederhole
    1. Links oben beginnen
    2. Wiederhole
        1. Artikel lesen
        2. Artikel bewerten
        3. Nächsten Artikel suchen
    3. Bis rechts unten angekommen
    4. Umblättern
3. Bis Zeitschrift zuende
4. Striche addieren
5. Wenn mehr als fünf Striche:
    1. Zeitschrift abonnieren

Beispiel-Algorithmus (3)

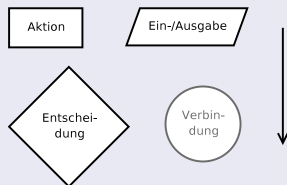
Artikelbewertung

1. Falls Wohnungsinserat und Wohnung in Innenstadt-Lage und *Miete* $<=$ € 500
   1. Artikel ausschneiden
   2. Artikel in Kartei einordnen
   3. Eintrag in Strichliste machen

Flussdiagramme

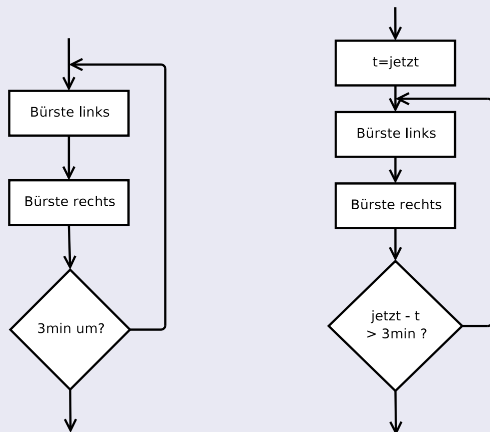
Ein Flussdiagramm ist eine grafische Darstellung eines Algorithmus, wobei verschiedene Symbole Aktionen, Entscheidungen, etc. symbolisieren. Von den etwa 30 definierten Symbolen sind nur die folgenden wirklich wichtig (auch die Verbindung wird häufig weggelassen):

Symbole



Ein einfaches Beispieldiagramm

Zähne putzen – zwei Versionen



Elemente von Computerprogrammen

Literale Direkt im Programm angegebene Werte (z.B. „3 Minuten“ im vorigen Flussdiagramm)

Konstante Feste Werte, die sich im Verlauf des Programms nicht ändern; benannt mit symbolischem Namen

Variable Speicher für veränderliche Werte, z.B. „t“ für die jeweilige Startzeit.

Dateien Datensammlungen, die gelesen und geschrieben werden. Das *Terminal* zur Benutzerinteraktion ist ebenfalls eine[3] Datei.

[3] Genaugenommen: drei verschiedene

Elemente von Computerprogrammen

Literale Direkt im Programm angegebene Werte (z.B. „3 Minuten“ im vorigen Flussdiagramm)

Konstante Feste Werte, die sich im Verlauf des Programms nicht ändern; benannt mit symbolischem Namen

Variable Speicher für veränderliche Werte, z.B. „t“ für die jeweilige Startzeit.

Dateien Datensammlungen, die gelesen und geschrieben werden. Das *Terminal* zur Benutzerinteraktion ist ebenfalls eine[3] Datei.

Anweisungen Führen allgemein Aktionen aus, die Daten verändern.

Kontrollstrukturen Steuern den Programmfluss in Abhängigkeit von bestimmten Daten (z.B. die Entscheidung „3 Minuten um?“)

[3] Genaugenommen: drei verschiedene

Allgemeines zur Sprache Perl

- 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 Schnitstellen zu anderen Programmen (z.B. Malaga)

Allgemeines zu Perl-Skripten

- Einfache Textdateien, üblicherweise mit der Endung „.pl“ (nur unter Windows notwendig bzw. dringend empfohlen – ansonsten muss der Perl-Interpreter immer explizit aufgerufen werden!)
- Vorsicht mit verschiedenen Zeilenumbrüchen (DOS vs. Unix), kann zu rätselhaften Fehlern führen

Aufruf-Alternativen auf der Kommandozeile

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

Hello, world!

Das erste Programm

```
my $hello = "Hello,␣world!";
print $hello . "\n";
```

Hello, world!

Das erste Programm

```
my $hello = "Hello,␣world!";
print $hello . "\n";
```

- Befehle werden mit Semikolon abgeschlossen.

Hello, world!

Das erste Programm

```
my $hello = "Hello, world!";
print $hello . "\n";
```

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

Hello, world!

Das erste Programm

```
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)
  - Einzeln: **my** $variable;

Hello, world!

Das erste Programm

```
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)
  - Einzeln: **my** $variable;
  - Mehrere: **my** ($var1, $var2, $var3);

Hello, world!

Das erste Programm

```
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)
  - Einzeln: **my** $variable;
  - Mehrere: **my** ($var1, $var2, $var3);
  - Initialisierung (=Wertzuweisung) schon bei der Deklaration möglich

Hello, world!

Das erste Programm

```
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)
  - Einzeln: **my** $variable;
  - Mehrere: **my** ($var1, $var2, $var3);
  - Initialisierung (=Wertzuweisung) schon bei der Deklaration möglich
- Erster Operator: Stringkonkatenation ($hello . "\n")

Hello, world!

Das erste Programm

```
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)
  - Einzeln: **my** $variable;
  - Mehrere: **my** ($var1, $var2, $var3);
  - Initialisierung (=Wertzuweisung) schon bei der Deklaration möglich
- Erster Operator: Stringkonkatenation ($hello . "\n")
- Eingebaute funktion `print` zur Textausgabe.

Variablen in Perl (1)

Perl ist eine *schwach typisierte Sprache* und kommt mit 3 bzw. 5 Datentypen aus, die sich an ihrem Namenspräfix unterscheiden lassen:

1. Skalare: Präfix „$“, z.B. „$myvariable“
   - Aus Sicht der Sprache atomarer Wert; jede Zuweisung ändert immer den kompletten Variableninhalt.
   - Speichert 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, die sich an ihrem Namenspräfix unterscheiden lassen:

1. Skalare: Präfix „$“, z.B. „$myvariable“
   - Aus Sicht der Sprache atomarer Wert; jede Zuweisung ändert immer den kompletten Variableninhalt.
   - Speichert Strings und Zahlenwerte (Referenzen: später)
   - Keine Unterscheidung zwischen Integer- und Fließkommazahlen, nicht einmal zwischen Zahlen und Strings (legal: **print** "3.5" * 3)

Skalarvariablen: Beispiele

```
my $s = "Hallo";     # Deklaration und  Initialisierung
my $t;               # Deklaration ohne Wertzuweisung
$s = 1;              # Selbe Variable  nimmt Zahlenwert auf
$t = 2.5;            # Jede Zahl  ist  Fliesskommazahl
$s = $t;             # Auch $s enthaelt  jetzt  den Wert 2.5
```

Variablen in Perl (2)

2. Listen (Arrays): Präfix „@“, z.B. „@mylist“
   - Eindimensionale Sammlungen beliebiger skalarer Werte
   - Einzelne Werte werden über Integer-Indizes in eckigen Klammern angesprochen, z.B. **print** $array[5]

Variablen in Perl (2)

2. Listen (Arrays): Präfix „@“, z.B. „@mylist“
    - Eindimensionale Sammlungen beliebiger skalarer Werte
    - Einzelne Werte werden über Integer-Indizes in eckigen Klammern angesprochen, z.B. **print** $array[5]
3. Hashes (assoziative Arrays): Präfix „%“, z.B. „%hash_map“: assoziieren String-Schlüssel mit beliebigen skalaren Werten (näheres später)

Variablen in Perl (3)

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 (3)

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 (3)

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 in zusammengesetzten Typen: entscheidend für den benutzten Präfix ist der Ziel-Typ!

$scalar = $array2[1]; →Einzelnes Element (Skalar!) kopieren
$array2[1] = "text";→Einzelnes Element überschreiben
@array1 = @array2; →Ganzes Array 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*[4]: Inhalt des Strings wird als Name eines Programms interpretiert und die Ausgabe dieses Programms eingefügt: `$vz = `ls -l`;`

[4] Auf der deutschen Tastatur links neben der Rücktaste, mit Shift, ggf. Leertaste hinterher.

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 „`dir C:\`“ befüllen (Vorsicht: der Backslash leitet Escape-Sequenzen ein und muss deshlab doppelt eingegeben werden!).
- Macht es einen Unterschied, ob Sie Ihre Variablen deklarieren oder nicht? Was ändert sich mit einem „**use** strict;“ vor dem Programmcode?
- Speichern Sie Ihren Vor- und Nachnamen in zwei oder mehr Stringvariablen und geben Sie sie konkateniert aus. Tun sie das selbe mit den Variablen in einen String interpoliert.

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 Ergbnis 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;

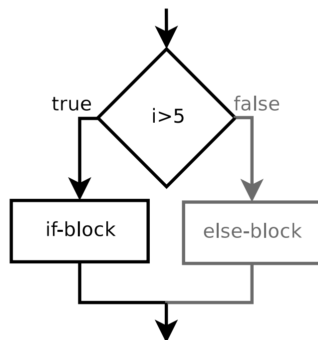
Übung

- Schreiben Sie ein Programm, das eine Anzahl Meilen einliest und sie in Kilometer umrechnet. Der Umrechnungsfaktor beträgt 1.609.
- Ändern Sie das Sekundenumrechnungsprogramm so, dass die Sekunden vom Terminal eingelesen werden.

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 ausgefuehrt werden
  # wenn $i > 5 ist
} else {
  # Befehle, die ausgefuehrt 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

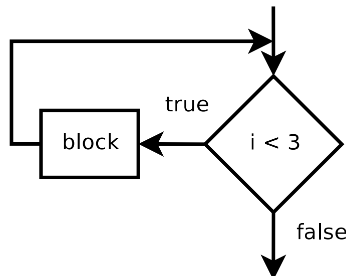
- String-Vergleich: `eq`, `ne`, `lt` und `gt`
  z.B: $lessThan = "Hund" lt "Katze";
- Vergleichsoperatoren zum sortieren: `<=>` (numerisch) und `cmp` (textuell)
- Logische Operatoren `!` (Negation, s. o.), `&&`/`and`: UND, `||`/`or`: ODER, `xor`: Exklusiv-ODER.
  - Achtung: nicht verwechseln mit Bit-Operatoren „&" 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 (Skript aus der Pseudodatei `NUL:` lesen lassen: `mein.pl <NUL:`) und berücksichtigen Sie diesen Fall bei der Verarbeitung der Eingabe.
- Fragen Sie vom Benutzer Namen, Alter und Größe ab, und begrüßen Sie ihn entsprechend. Versuchen Sie, unrealistische Angaben in der Ausgabe zu kommentieren.
- Schlagen Sie in der Dokumentation die Funktion **length**() nach und benutzen Sie sie im vorigen Programm.

Mehr Kontrollstrukturen: `while` und `do-while`

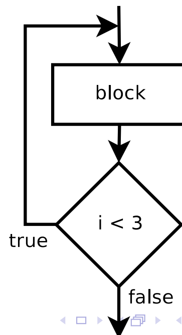
```
while($i < 3) {
   # Befehle, die ausgefuehrt werden
   # solange $i < 3 ist
}
```



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
F	F	F
F	W	F
W	F	F
W	W	W

a	b	a OR b
F	F	F
F	W	W
W	F	W
W	W	W

a	b	a XOR b
F	F	F
F	W	W
W	F	W
W	W	F

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

Wahrheitstabellen logischer Operatoren:

a	b	a AND b
F	F	F
F	W	F
W	F	F
W	W	W

a	b	a OR b
F	F	F
F	W	W
W	F	W
W	W	W

a	b	a XOR b
F	F	F
F	W	W
W	F	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.

Listen (1)

- Listenindizes zählen von null ab, also $liste[0], $liste[1], ...
- Deklaration und Initialisierung einer Liste:
  my @zeug = ("Laptop", "Zettel", "Stift");
- Solange die Elemente skalar sind, sind ihre genaueren Eigenschaften egal:
  my @mischmasch = (0, "Liste", "", 5.3e5, "Ende");

Listen (1)

- Listenindizes zählen von null ab, also $liste[0], $liste[1], ...
- Deklaration und Initialisierung einer Liste:
  my @zeug = ("Laptop", "Zettel", "Stift");
- Solange die Elemente skalar sind, sind ihre genaueren Eigenschaften egal:
  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 (1)

- Listenindizes zählen von null ab, also $liste[0], $liste[1], ...
- Deklaration und Initialisierung einer Liste:
  my @zeug = ("Laptop", "Zettel", "Stift");
- Solange die Elemente skalar sind, sind ihre genaueren Eigenschaften egal:
  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 (1)

- Listenindizes zählen von null ab, also $liste[0], $liste[1], ...
- Deklaration und Initialisierung einer Liste:
  my @zeug = ("Laptop", "Zettel", "Stift");
- Solange die Elemente skalar sind, sind ihre genaueren Eigenschaften egal:
  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;
```

- Interpolation wie bei Skalaren, z. B.: **print** "Liste:␣@list\n";

Listen (2)

- Teillisten (so) nicht möglich:
  (1, 2, (3, 4), 5) ist das selbe wie (1, 2, 3, 4, 5)
  Grund: Listen werden ineinander *interpoliert* (hier: „flachgeklopft" – Listen sind immer eindimensional. Nicht verwechseln mit der String-Interpolation!).

Listen (2)

- Teillisten (so) nicht möglich:
  (1, 2, (3, 4), 5) ist das selbe wie (1, 2, 3, 4, 5)
  Grund: Listen werden ineinander *interpoliert* (hier: „flachgeklopft" – Listen sind immer eindimensional. Nicht verwechseln mit der String-Interpolation!).
- Bequeme Schreibweise für einfache Listenelemente: **qw**() („quote words"):
  @tiere = **qw**(Hund Katze Maus Ameisenbaer);

Listenmanipulation

- Am Listenende: ←push / pop →
- Am Listenanfang: ←shift / unshift →

Listenoperationen

```
my (@list,$elem);

# $elem am Ende anhaengen
push @list, $elem;

# Letztes Element der Liste entfernen und in $elem kopieren
$elem = pop @list;

# Erstes Element der Liste entfernen und in $elem kopieren
$elem = shift @list;

# $elem am Anfang einfuegen
unshift @list, $elem;
```

Kontext

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

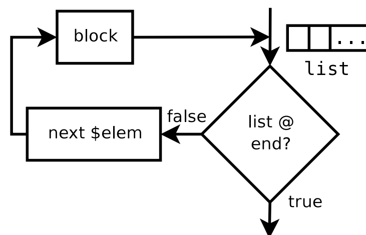
1. Skalarkontext
   - Z.B.: $var = @list; oder **print** @list . `"n"`;
   - Unterteilt in String-, numerischen und „egal“-Kontext (meist irrelevant)
   - Kann erzwungen werden: **print scalar**(@list);
2. Listenkontext
   - Z.B. **print***Ausdruck*; (**print** verlangt eine Liste);
     `($bla, Ausdruck, $fasel);` (Interpolation von *Ausdruck*)
     @lines = ‘dir‘; (Zuweisung an eine Liste)
   - Wenn eine Funktion ihre Argumente im Listenkontext auswertet, ist dies 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?

Listeniteration mit foreach

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

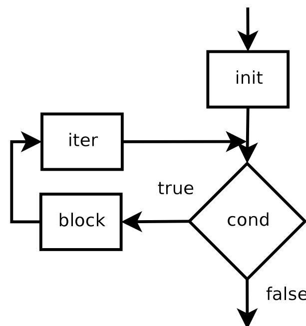


- Anweisungen werden für jedes Element in @list wiederholt, wobei $scalar auf das jeweilige Element verweist.
- Achtung: $scalar ist *keine Kopie* sondern ein sog. *Alias* des jeweiligen Listenelements! D. h. jede Modifikation desselben verändert @liste selbst.

Zählschleifen mit `for`

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

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



- Die Notation --$i ist Kurzschreibweise für $i=$i−1. Für andere In-/Dekremente gibt es auch z. B. $i += 2 (geht auch mit anderen Operatoren!)
- „init“, „cond“ und „iter“ können beinahe beliebige Anweisungen enthalten oder auch ganz entfallen. Die Semikolons sind obligatorisch.

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);
    }
}
```

Übung

- Laden Sie die Datei unter `http://www.linguistik.uni-erlangen.de/~msbethke/teaching/SS2007-PerlLing/limas.txt` herunter und lesen Sie sie zeilenweise ein (Umleitung der Eingabe aus einer Datei wie gehabt: `meinskript.pl < limas.txt`)
- Schlagen Sie in der `perlfunc`-Dokumentation die Funktion **split** nach und verwenden Sie sie, um die Zeilen in Wörter aufzuteilen.
- Geben Sie die Datei wortweise wieder aus.
- Zählen Sie die Vorkommen des Wortes „das“ in der Datei.

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 Hochkommas (`'`)!

Reguläre Ausdrücke: Literale und Zeichenklassen

- Buchstaben, Ziffern und viele Sonderzeichen stehen in RA für sich selbst; der RA „`a`“ steht also einfach für ein „a“.
- Das selbe gilt für Aneinanderreihungen derselben; der RA „`hallo`“ steht also für eben dieses Wort.

Reguläre Ausdrücke: Literale und Zeichenklassen

- Buchstaben, Ziffern und viele Sonderzeichen stehen in RA für sich selbst; der RA „a“ steht also einfach für ein „a“.
- Das selbe gilt für Aneinanderreihungen derselben; der RA „`hallo`“ steht also für eben dieses Wort.
- Eine *Zeichenklasse* in eckigen Klammern steht für *ein* beliebiges in der Klasse enthaltenes Zeichen; der RA „`[Abc]`“ steht also für ein großes „A“ oder ein kleines „b“ oder „c“.
- Zeichenklassen können Zeichen explizit auflisten oder *Bereiche* definieren: „`[A-Z0-9]` steht z. B. für alle Großbuchstaben und Ziffern.
- Eine eckige Klammer kann als erstes Element in eine Zeichenklasse aufgenommen werden, ein Bindestrich als erstes oder letztes. Zum Beispiel: `[]a-z-]`
- Ist das erste Zeichen der Klasse ein Zirkumflex oder *Caret*, wird die Klasse negiert und steht für irgendein *nicht* enthaltenes Zeichen. Zum Beispiel: `[^aeiou]`

Reguläre Ausdrücke: Spezialklassen

- Der Punkt steht für irgendein beliebiges Zeichen
- Der POSIX-Standard vergibt für häufig gebrauchte Zeichenklassen symbolische Namen, die wie ein Zeichen in einer Klasse gebraucht werden können. Die wichtigsten:

  `[:lower:]` Kleinbuchstaben – was als Buchstabe gilt, ist locale-abhängig!

  `[:upper:]` Großbuchstaben, dito

  `[:alpha:]` Buchstaben: `[:lower:]` und `[:upper:]`

  `[:digit:]` Ziffern: 0, 1, 2,... bis 9.

  `[:alnum:]` Alphanumerische Zeichen: `[:alpha:]` und `[:digit:]`

  `[:blank:]` Leerzeichen und Tabulator.

  `[:space:]` Whitespace: `[:blank:]` sowie vertikaler Tabulator, Zeilen- und Seitenvorschub, Wagenrücklauf.

  `[:punct:]` Interpunktionszeichen: `,`, `-`, `!`, `"`, `#`, `$`, usw.

Reguläre Ausdrücke: Quantoren (1)

Quantoren geben an, wie oft das vorangehende *Atom*[5] in einem Ausdruck erscheinen darf.

- Der Stern oder *Kleene-Operator* bedeutet dabei „beliebig oft“, d. h. auch null-mal. So passt der Ausdruck „`ab.*`“ auf beliebig lange Zeichenketten, die mit „ab“ anfangen, auch „ab“ selbst.

[5] Ein Atom ist entweder ein einzelnes Zeichen, eine Zeichenklasse oder ein Klammerausdruck.

Reguläre Ausdrücke: Quantoren (1)

Quantoren geben an, wie oft das vorangehende *Atom*[5] in einem Ausdruck erscheinen darf.

- Der Stern oder *Kleene-Operator* bedeutet dabei „beliebig oft", d. h. auch null-mal. So passt der Ausdruck „`ab.*`" auf beliebig lange Zeichenketten, die mit „ab" anfangen, auch „ab" selbst.
- Das Pluszeichen steht für „beliebig oft, aber mindestens einmal". Die Ausdrücke „`a+`" und „`aa*`" sind also äquivalent.

[5] Ein Atom ist entweder ein einzelnes Zeichen, eine Zeichenklasse oder ein Klammerausdruck.

Reguläre Ausdrücke: Quantoren (1)

Quantoren geben an, wie oft das vorangehende *Atom*[5] in einem Ausdruck erscheinen darf.

- Der Stern oder *Kleene-Operator* bedeutet dabei „beliebig oft", d. h. auch null-mal. So passt der Ausdruck „`ab.*`" auf beliebig lange Zeichenketten, die mit „ab" anfangen, auch „ab" selbst.
- Das Pluszeichen steht für „beliebig oft, aber mindestens einmal". Die Ausdrücke „`a+`" und „`aa*`" sind also äquivalent.
- Das Fragezeichen bedeutet „optional" bzw. „null- oder einmal".

[5] Ein Atom ist entweder ein einzelnes Zeichen, eine Zeichenklasse oder ein Klammerausdruck.

Reguläre Ausdrücke: Quantoren (1)

Quantoren geben an, wie oft das vorangehende *Atom*[5] in einem Ausdruck erscheinen darf.

- Der Stern oder *Kleene-Operator* bedeutet dabei „beliebig oft“, d. h. auch null-mal. So passt der Ausdruck „`ab.*`“ auf beliebig lange Zeichenketten, die mit „ab“ anfangen, auch „ab“ selbst.
- Das Pluszeichen steht für „beliebig oft, aber mindestens einmal“. Die Ausdrücke „`a+`“ und „`aa*`“ sind also äquivalent.
- Das Fragezeichen bedeutet „optional“ bzw. „null- oder einmal“.
- Vorsicht: der Ausdruck vor dem Quantor wird bei jedem erneuten Abpassen neu ausgewertet!

[5] Ein Atom ist entweder ein einzelnes Zeichen, eine Zeichenklasse oder ein Klammerausdruck.

Reguläre Ausdrücke: Quantoren (2)

- Numerische Quantoren bestehen aus einer Zahl oder einem mit Komma getrennten Zahlenpaar in geschweiften Klammern.
  - Eine einzelne Zahl steht für „genau so oft“, z. B. „`a{5}`“ für „genau 5 aufeinanderfolgende a“.
  - Ein Zahlenpaar gibt die minimale und maximale Anzahl an, z. B. passt „`S[ea]{2,3}.+`“ auf „Seeadler“ und „Seattle“, nicht aber „Sessel“.
  - Eine der Minimum- bzw. Maximum-Angaben kann jeweils entfallen[6] wobei dann null respektive unendlich angenommen wird:
    `ab{3,}`: ein a gefolgt von mindestens 3 b
    `b.{,10}`: ein b gefolgt von höchstens 10 beliebigen Zeichen

[6] Bei *egrep* nur das Maximum, d.h. `{2,}` funktioniert, `{,2}` aber nicht!

Reguläre Ausdrücke: Gruppierung und Alternation

- Mit runden Klammern können beliebig lange Teile eines RA zu *Atomen* zusammengefasst werden.

[7] Wo dies nicht erwünscht ist, lässt es sich manchmal abschalten, z. B. in *PerlREs* mit `(?:...)`, in *vim* mit `\%(...)`.

Reguläre Ausdrücke: Gruppierung und Alternation

- Mit runden Klammern können beliebig lange Teile eines RA zu *Atomen* zusammengefasst werden.
  - Erlaubt die Anwendung von Quantoren auf den gesamten geklammerten Ausdruck.
  - Stellt i. d. R. den Text, auf den der Klammerausdruck passt, in einem speziellen Register zur Verfügung[7], wo er z. B. in Ersetzungstexten benutzt werden kann.
- Beispiel: „`(hallo){,2}`“ passt auf die leere Zeichenkette(!), „hallo“ und „hallohallo“.

[7]Wo dies nicht erwünscht ist, lässt es sich manchmal abschalten, z. B. in *PerlREs* mit `(?:...)`, in *vim* mit `\%(...)`.

Reguläre Ausdrücke: Gruppierung und Alternation

- Mit runden Klammern können beliebig lange Teile eines RA zu *Atomen* zusammengefasst werden.
  - Erlaubt die Anwendung von Quantoren auf den gesamten geklammerten Ausdruck.
  - Stellt i. d. R. den Text, auf den der Klammerausdruck passt, in einem speziellen Register zur Verfügung[7], wo er z. B. in Ersetzungstexten benutzt werden kann.
- Beispiel: „`(hallo){,2}`“ passt auf die leere Zeichenkette(!), „hallo“ und „hallohallo“.
- Der senkrechte Strich (→„Pipe-Symbol“) markiert eine Alternation und wird üblicherweise mit Klammern benutzt, z. B. `x(ABC|abc)y` für „xABCy“ oder „xabcy“, nicht aber „xAbcy“ (vgl. „`x[AaBbCc]{3}y`“!).

[7] Wo dies nicht erwünscht ist, lässt es sich manchmal abschalten, z. B. in *PerlREs* mit `(?:...)`, in *vim* mit `\%(...)`.

Weitere Sonderzeichen

Einige Muster repräsentieren abstrakte Einheiten im Text wie z. B. Wortgrenzen. Für diese existieren Sonderzeichen oder sog. *Escape-Sequenzen*, d. h. Kombinationen eines Backslash mit einem anderen Zeichen.

- `^` Zeilenanfang
- `$` Zeilenende
- `\b` Wortzwischenraum (Anfang oder Ende, auch bei Satzzeichen)
- `\B` Alles außer Wortzwischenräumen
- `\<` Zwischenraum am Wortanfang
- `\>` Zwischenraum am Wortende

- `\w` *Word character*, entspricht `[[:alnum:]]`
- `\W` *Non-word character*, entspricht `[^[:alnum:]]`

Reguläre Ausdrücke (2): Match / Bindung

- `m//`: „Match“-Operator, testet auf Übereinstimmung mit einem RA.
  - `m/fo{2,}bar/;`
  - `m!^/usr/bin/!;`
  - 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/!;`
  - Werden Schrägstriche verwendet, kann das ‚m' wegfallen: `while(<>) { print unless /^#/; }`
- `=~`: Bindungsoperator, bindet eine Variable zur Bearbeitung an einen RA. Z.B.: `$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
  - `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/&fasel/
  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 (Ausdruck rechts!)