Angewandtes Programmieren
PERL für Fortgeschrittene

Dr. Helmut Schmid

Sommersemester 2014

Angewandtes Programmieren
PERL für Fortgeschrittene

Dr. Helmut Schmid

Skript zur Vorlesung
Centrum für Informations- und Sprachverarbeitung
Ludwig–Maximilians–Universität
München

Basierend auf einer Vorlage von Holger Bosk und Dr. Michaela Geierhos

Inhaltsverzeichnis

1 Subroutinen

1.1 Vorbemerkung

Um große Programmieraufgaben übersichtlich und strukturiert zu realisieren, ist es notwendig, bestimmte Teilaufgaben eines Programms in Unterprogrammen umzusetzen. Diese können unabhängig entwickelt, getestet, korrigiert, optimiert, und in anderen Programmen wiederverwendet werden. Diese Technik der Programmierung nennt man *modulare Programmierung*: Eigenständige Unteraufgaben werden getrennt in Modulen erledigt. Eine Programmiersprache, die diese Technik unterstützt, muss in der Lage sein,

- Programmteile in Blöcken zusammenzufassen,
- ihnen einen Namen zu geben,
- Werte aus anderen Programmteilen an sie weiterzureichen,
- Parameter innerhalb dieser Sequenzen neu zu berechnen,
- und sie anschließend an das aufrufende Programm zurückzugeben.

PERL unterstützt die *modulare Programmierung*: Programmteile können als Subroutine zusammengefasst werden und dieser können wiederum Werte übergeben werden. Auch können von Subroutinen Werte berechnet werden, die dem aufrufenden Programm später zurückgegeben werden und so im aufrufenden Programm wieder zur Verfügung stehen. Somit sind Subroutinen wiederverwendbare Code-Blöcke. Sie entsprechen den Funktionen, Methoden und Prozeduren anderer Programmiersprachen.

1.2 Eigenschaften von Subroutinen

- Subroutinen können überall im Programm deklariert werden
- Subroutinen akzeptieren eine variable Anzahl von Parametern
- Subroutinen können mehrere Rückgabewerte haben
- Subroutinen haben schwach typisierte Parameter- und Rückgabewerte
- Subroutinen sind rekursiv aufrufbar

1.3 Definition von Subroutinen

Wir unterscheiden Subroutinen, die man selbst definiert, und solche, die PERL zur Verfügung stellt. Erstere werden als *benutzerdefiniert*, während letztere als *built-in* bezeichnet werden.

Definition mit sub

```
1  sub my_sub {
2
3      <Programmcode>
4
5  }
```

Wie bereits erwähnt wurde, können Subroutinen überall dort im Programm deklariert werden, wo auch normale Anweisungen stehen können. Außerdem ist es möglich, Subroutinen zur Laufzeit – *on the fly* – zu deklarieren.

1.4 Aufruf von Subroutinen

Subroutinen können mit oder ohne Argumente aufgerufen werden:

- Übergabe mit Argumenten: `my_subroutine(arg1, ..., argn);`
- oder: `&my_subroutine(arg1, ..., argn);`
- falls keine Argumente übergeben werden sollen: `my_subroutine();`
- oder: `&my_subroutine();`

1.5 Auslesen der Argumente in der Subroutine

Alle Argumente, die beim Aufruf einer Subroutine übergeben werden, sind in der Subroutine als Elemente von `@_` zugänglich. Typischerweise kopiert man als erstes in der Subroutine den Inhalt von `@_` in lokale Variablen. Der Subroutinenblock, der mit geschweiften Klammern umschlossen wird, besteht dann aus einer Parametersektion und einem Anweisungsblock.

In der **Parametersektion** werden die Parameter der Subroutine definiert. Die Parameter stellen die Schnittstelle zwischen dem aufrufenden Programm und dem Unterprogramm dar und nehmen die Werte beim Aufruf der Subroutine an. Damit haben sie alle notwendigen Informationen, um die übergebenen Werte weiter zu verarbeiten.

Im **Anweisungsblock** stehen alle Anweisungen, die ausgeführt werden, wenn die Subroutine aufgerufen wird. Die übergebenen Parameter der Subroutine fungieren als lokale Variablen in diesem Unterprogramm; ihre Startwerte haben sie beim Aufruf bekommen.

Da die Programmiersprache PERL beim Aufruf einer Subroutine nicht die Anzahl und die Typen der übergebenen Parameter überprüft, muss der Programmierer unbedingt selbst darauf achten, dass es hier keine Widersprüche gibt. Dies ist generell das empfehlenswerteste Verfahren:

```
1  #!/usr/bin/perl
2
3  use strict;
4  use warnings;
5
6  { # main program
7    my $first_side;
8    my $second_side;
9    my $third_side;
10
11   print "\nPlease enter the length of the first adjacent side of
         the triangle: ";
12   $first_side = <>;
13   print "\nPlease enter the length of the second adjacent side of
         the triangle: ";
```

```
14    $second_side = <>;
15
16    $third_side = hypotenuse($first_side, $second_side);
17    print "\nThe length of the third side is $third_side\n\n";
18  }
19
20  # the subroutine:
21  sub hypotenuse {
22    my ($side1, $side2) = @_;
23    return sqrt ( ($side1 ** 2) + ($side2 ** 2) );
24  }
```

Alternativen zur Zeile `my ($side1, $side2)= @_`:

```
1  my $side1 = $_[0];
2  my $side2 = $_[1];
```

oder auch:

```
1  my $side1 = shift;
2  my $side2 = shift;
```

(wenn shift ohne Argument aufgerufen wird, bezieht es sich automatisch auf `@_`).

Da alle Argumente der Subroutine sich in dem einzigen Array `@_` wiederfinden, ist es z.B. bei einem Aufruf `$result = my_sub(@array1, @array2)` nicht möglich, innerhalb der Subroutine zu ersehen, welches Element von `@_` zu `@array1` gehörte, und welches zu `@array2`. Dieses Problem lässt sich aber relativ einfach dadurch lösen, dass man Referenzen auf Arrays übergibt statt die Arrays selbst.

1.6 Rückgabe eines Ergebniswertes aus einer Subroutine

Üblicherweise erfolgt die Rückgabe der Ergebnisse einer Subroutine mit einer `return`-Anweisung (siehe Beispielprogramm zum Auslesen der Argumente).

Wie auch bei der Übergabe von Argumenten an eine Subroutine werden die Rückgabewerte einer Subroutine (Listenkontext im aufrufenden Programm vorausgesetzt) alle als Elemente einer einzigen *flachen* Liste zurückgegeben. Bei `return @array1, @array2;` ist im aufrufenden Programm nicht mehr ersichtlich, welches Element zu welchem Array gehörte, alles steht jetzt in einer Liste.
Wenn keine `return`-Anweisung vorhanden ist, wird der Wert des zuletzt ausgewerteten Ausdrucks zurückgegeben.

Aufruf von return ohne Argumente

Dies liefert im aufrufenden Programm im Listenkontext eine leere Liste zurück, jedoch im skalaren Kontext ein `undef`. In beiden Fällen ist dieses Ergebnis logisch gesehen *falsch* – der Aufruf von `return` ohne Argumente eignet sich daher sehr gut, um ggfs. die Information über eine nicht erfolgreiche Ausführung der Subroutine an das Hauptprogramm zu übermitteln.
Man kann also in der Subroutine so etwas schreiben wie:

```
1  if ($success) {
2    return $return_value;
3  } else {
4    return;
5  }
```

Im aufrufenden Programm kann man dann wie folgt das Ergebnis der Subroutine abfragen:

```
1  if ($result = my_sub($var1, $var2)) {
2    print "The result is $result\n";
3  }
4  else {
```

```
5    print "The subroutine was not successfull\n";
6    # oder andere Reaktion als Folge der fehlgeschlagenen Subroutine
7  }
```

Dies funktioniert auch im Listenkontext:

```
1  if (@result_array = my_sub($var1, $var2)) {
```

Dabei impliziert die Kontrollabfrage `if ($result = my_sub($var1, $var2))` allerdings, dass auch andere logisch *falsche* Werte (z.B. eine Null oder ein leerer String) als Fehlschlagen der Subroutine gedeutet werden. Will man dies vermeiden, muss man das Ergebnis explizit mit `defined` abfragen. Eine Alternative zum Kopieren der übergebenen Argumente in eigene Variablen und zur expliziten Rückgabe von Werten ist das Auslesen von `@_` mit impliziter Rückgabe von Werten.
(**Vorsicht!** Diese Methode verändert die Werte im aufrufenden Programm!)

Beispiel:

```
1  #!/usr/bin/perl
2
3  use strict;
4  {
5    my @list;
6    @list=(5,7,9);
7    increment(@list);
8    print join(' ', @list), "\n";
9  }
10
11 sub increment {
12   foreach my $elem (@_) {
13     $elem++;
14   }
15 }
```

Diese Methode hat einerseits einige Vorteile: man spart sich die explizite Abfrage eines Rückgabewertes, und in der Subroutine spart man sich das Kopieren der Argumente in eigene Variablen.

Dennoch ist diese Methode im allgemeinen NICHT empfehlenswert: Variablen werden im aufrufenden Programm in nicht erkennbarer Weise verändert; eine Zuweisung wie `@incremented_list = increment(@list);` hingegen macht das Verhalten des Programms und die Belegung der Variablen sehr viel durchschaubarer. Wir werden daher im Kurs ohne diese direkte Manipulation von `@_` arbeiten und stattdessen immer explizit einen Ergebniswert mit `return` zurückgeben.

1.7 Übungsaufgaben

Aufgabe 1.1 Schreibe eine Subroutine, die alle Werte eines Arrays mit 2 multipliziert. Im Hauptprogramm soll ein Array angelegt werden, dann soll die Subroutine aufgerufen werden und dann soll der Inhalt des Arrays ausgegeben werden. Das Hauptprogramm (ohne die Subroutine) soll dabei in geschweiften Klammern stehen.

(a) Schreibe das Programm so, dass die Subroutine direkt über `@_` die Werte im Hauptprogramm verändert.

(b) Schreibe eine zweite Variante, bei der im Hauptprogramm ein Array an die Subroutine übergeben wird; die Subroutine gibt dann ein verändertes Array an das Hauptprogramm zurück. Es soll dabei in der Subroutine keine direkte Manipulation der Werte im Hauptprogramm mittels `@_` geben.

2 Referenzen auf Variablen

2.1 Was sind Referenzen?

Referenzen sind so etwas wie *Zeiger* auf andere Variablen. So kann man z.B. ein Array in einem Programm anlegen und dann einen Zeiger darauf:

```
1 @mein_array = (3,5,7); # Anlegen des Arrays
2 $array_referenz = \@mein_array;
3 # Erzeugen einer Referenz auf das Array
```

`$array_referenz` fungiert jetzt als *Zeiger* auf `@mein_array`, d.h. man kann jederzeit die Daten in `@mein_array` mittels `$array_referenz` ansprechen (wie das genau aussieht, kommt noch).

2.2 Wie erzeugt man Referenzen?

Wie im obigen Beispiel kann man Referenzen generell durch das Voranstellen eines Backslashs vor eine Variable erzeugen:

```
1 $scalar_reference = \$var;
2 $array_reference = \@array;
3 $hash_reference = \%hash;
4 $code_reference = \&subroutine;
```

2.3 Und wie ruft man die Inhalte wieder ab?

Das Prinzip ist ganz einfach: Referenzen kann man überall dort einsetzen, wo man sonst den Namen einer Variablen schreiben würde:

```
1 # Anlegen des Arrays
2 @mein_array = (3,5,7);
3 # Erzeugen einer Referenz auf das Array
4 $array_referenz = \@mein_array;
5 # Abrufen der Inhalte des Arrays, "Dereferenzieren"
6 foreach $zahl (@$array_referenz) {
7   print "$zahl\n";
8 }
```

In Zeile 3 würde man normalerweise `(@mein_array)` schreiben; hier ersetzen wir den Namensteil der Variablen `mein_array` durch die Referenz (inklusive vorangestelltem `$`), so dass sich `(@$array_referenz)` ergibt. Der Effekt von beidem ist genau gleich, d.h. das Programm funktioniert genauso, egal ob man `@mein_array` schreibt, oder ob man den Umweg über die Referenz wählt und `@$array_referenz` schreibt.
Statt `@$array_referenz` kann man auch noch expliziter `@{$array_referenz}` schreiben.

Entsprechend auch bei Hashes: Wo man ohne Referenzen schreibt:

```
1 %hash = (hallo => 7, haus => 19);
2 foreach $key (keys%hash) {
3 ....
```

würde man mit Referenzen schreiben:

```
1 $hash_ref = \%hash;
2 foreach $key (keys%$hash_ref) {
3 # oder: foreach $key (keys%{$hash_ref}) {
4  ....
```

Abfrage von einzelnen Array- und Hash-Elementen über Referenzen

Der Zugriff auf einzelne Elemente von Arrays oder Hashes folgt ebenfalls den oben genannten Regeln, so dass man wieder statt des Variablennamens die entsprechende Referenz einsetzt. Statt

```
1 @a = (1, 3, 5, 7, 9);
2 foreach $index (0..4) {
3   print $a[$index], "\n";
4 }
```

könnte man auch schreiben:

```
1 @a = (1, 3, 5, 7, 9);
2 $aref = \@a;
3 foreach $index (0..4) {
4  print ${$aref}[$index], "\n";
5 }
```

2.4 Referenzen als Schleifenvariablen

Ein Vorteil von Referenzen besteht darin, dass man sie hintereinander auf verschiedene Variablen zeigen lassen kann:

```
1  @array1 = (3,5,7);
2  @array2 = (4,6,8);
3  @array3 = (9,7,2,6);
4  @array4 = (12,1,23,4);
5
6  $array_ref = \@array1;
7  print @$array_ref;
8  $array_ref = \@array2;
9  print @$array_ref;
10 $array_ref = \@array3;
11 print @$array_ref;
12 $array_ref = \@array4;
13 print @$array_ref;
```

Hier kann man denselben `print`-Befehl verwenden, um zwei verschiedene Arrays auszudrucken. Mittels einer Schleife könnte das dann so aussehen:

```
1 @array1 = (3,5,7);
2 @array2 = (4,6,8);
3 @array3 = (9,7,2,6);
4 @array4 = (12,1,23,4);
5
6 foreach $array_ref (\@array1, \@array2, \@array3, \@array4) {
7     print @$array_ref, "\n";
8 }
```

Ohne Referenzen wäre es nicht möglich, die Schleifenvariable nacheinander die Werte verschiedener Arrays annehmen zu lassen – denn die Schleifenvariable muss stets eine skalare Variable sein. Mittels Referenzen ist es hier möglich, die kompletten Arrays jeweils mittels einer skalaren Variable anzusprechen.
Dasselbe funktioniert natürlich auch mit Hashes (nur dass die anders ausgegeben werden müssen).
(**Anmerkung:** Natürlich sollte man Arrays normalerweise etwas schöner ausgeben, z.B. mit `print join(", ", @array);`)

2.5 Der Befehl `ref`

Sollte man an einer Stelle im Programm nachprüfen wollen, ob eine skalare Variable eine einfache skalare Variable oder aber eine Referenz ist, kann man den `ref`-Befehl benutzen. Handelt es sich um eine Referenz, so gibt mir der `ref`-Befehl aus, auf welchen Datentyp die Referenz zeigt. Handelt es sich nicht um eine Referenz, gibt der `ref`-Befehl ein logisches *Falsch* zurück, was sich in der Ausgabe in einem leeren String äußert.

Beispiel:

```
1 $var = 7;
2 print ref($var), "\n";
3 @array = (7,4,6);
4  $var = \@array;
5 print ref($var), "\n";
```

Ausgabe: eine Leerzeile, dann eine Zeile mit dem Inhalt `ARRAY`.

2.6 Die Pfeil-Schreibweise

Eine andere Möglichkeit der Schreibweise ist der Pfeil. Mit ihm sagt man sinngemäß: „dereferenziere das, was links vom Pfeil steht und lies dann weiter nach rechts". D.h. nach

```
1 $hash{Buch} = "lesen";
2 $hash_ref = \%hash;
```

kann man auf den Schlüssel „Buch" mit folgenden Varianten zugreifen:

```
1 print ${$hash_ref}{Buch};
```

oder auch:

```
1 print $hash_ref->{Buch};
```

oder natürlich:

```
1 print $hash{Buch};
```

Die Ausgabe des Arrays aus dem obigen Beispiel würde mit der Pfeil-Schreibweise dann so aussehen:

```
1 @a = (1, 3, 5, 7, 9);
2 $aref = \@a;
3 foreach $index (0..4) {
4   print $aref->[$index], "\n";
5 }
```

2.7 Vorsicht, Mehrdeutigkeiten!

Wie bereits erwähnt, kann man der Eindeutigkeit halber beim Einsetzen von Referenzen an die Stelle von Variablennamen auch geschweifte Klammern um die Referenz setzen: `@{$array_referenz}` statt

einfach `@$array_referenz`. Besonders in komplexeren Fällen vermeidet die Schreibweise mit Klammern Zweideutigkeiten, z.B. beim Abfragen eines Hash-Wertes, wenn man den Hash über eine Referenz anspricht:

```
1 $$x{hallo} = 7;
```

könnte einmal bedeuten:

```
1 ${$x}{hallo} = 7;
2 # von dem Hash, auf den die Referenz $x zeigt, wird der Key "hallo"
      angesprochen und diesem der Wert 7 zugeordnet. Dies entspricht
     $x->{hallo} = 7;
```

oder aber:

```
1 ${$x{hallo}} = 7;
2 # es gibt einen Hash %x, in dem der Wert zu dem Key "hallo" eine
     Referenz auf eine skalare Variable ist, und dieser skalaren
     Variable soll der Wert 7 zugewiesen werden.
```

In vielen Fällen wird der Compiler auch ohne Klammern oder Pfeile einfach das interpretieren, von dem er meint, dass es wohl beabsichtigt war – in anderen Fällen wird er die Kompilierung mit einer Fehlermeldung abbrechen. Klammern zu setzen bzw. Pfeile zu benutzen vermeidet in jedem Fall mögliche Interpretationsprobleme.

2.8 Übungsaufgaben

Aufgabe 2.1 Weise einer skalaren Variablen einen Wert zu, erzeuge eine Referenz auf diese Variable und gib die Variable mit Hilfe der Referenz aus.

Aufgabe 2.2 Erzeuge eine skalare Variable, ein Array und ein Hash und dann auf jede der 3 Variablen eine Referenz. Gib aus, was ref(<Referenz>) für jede einzelne der 3 Referenzen ergibt.

Aufgabe 2.3 Erzeuge 3 Hashes, dann auf jedes der Hashes eine Referenz und gib mittels einer `foreach`-Schleife die Inhalte aus, wobei die Schleifenvariable über die Hash-Referenzen iteriert (analog zum Beispiel mit den 4 Arrays).

Aufgabe 2.4 Gegeben sei folgender Programmanfang:

```
1 #!/usr/bin/perl
2
3 use strict;
4 use warnings;
5
6 {
7   my @array1 = (3,5,7,8);
8   my @array2 = (4,5,6);
9 ....
```

Schreibe eine Subroutine, die mit einem einzigen Aufruf ausgibt, welche Elemente im Hauptprogramm in `@array1`, und welche in `@array2` gespeichert waren. Das Hauptprogramm (ohne die Subroutine!) soll dabei in geschweiften Klammern stehen.
(**Hinweis:** die Subroutine sollte bei ihrem Aufruf 2 Array-Referenzen als Argumente bekommen)

3

Übergabearten „Call by Value“ und „Call by Reference“

Über Rückgabewerte und Argumente von Subroutinen werden zwischen dem aufrufenden Programm und dem Unterprogramm Daten ausgetauscht. In der Programmierung unterscheidet man deshalb zwei Übergabearten von Daten:

1. Das Unterprogramm soll von einer Variablen, die im aufrufenden Programm definiert ist, nur den Wert lesen. Diese Übergabeart heißt „Call by Value“.

2. Das Unterprogramm darf auf eine Variable, die im aufrufenden Programm definiert ist, selbst zugreifen. Somit kann das Unterprogramm den Wert des Arguments lesen und im aufrufenden Programm auch verändern. Diese Übergabeart heißt „Call by Reference“.

Call by Value

Intern wird diese Übergabeart dadurch realisiert, dass dem Unterprogramm nur eine Kopie des Argumentwerts übergeben wird. Das Unterprogramm kann den Wert nur lesen. Der Wert der Variable im aufrufenden Programm kann nicht verändert werden. ⇒ Lesender Zugriff auf das Argument.

Beispiel:

```
1 my $text = "Hallo";
2 print ($text)
```

Der Subroutine `print` wird eine Kopie des Argumentwerts der Variable `$text` übergeben, und sie kann ihn auf dem Terminal ausgeben.

Call by Reference

Intern wird diese Übergabeart dadurch realisiert, dass der Programmierer dem Unterprogramm die Adresse der Argumentvariablen im aufrufenden Programm übergibt. Nun weiß das Unterprogramm, wo die Argumentvariable gespeichert ist und kann über den Zugriff auf die Adresse der Argumentvariable den Wert lesen, aber gleichzeitig auch verändern. ⇒ Lesender und schreibender Zugriff über die Adresse der Argumentvariable.

Damit in PERL die Adresse einer Variablen an ein Unterprogramm übergeben wird, muss man dem Variablennamen des Arguments beim Aufruf einen Backslash voranstellen. Dann wird die Adresse der Argumentvariablen übergeben.

Beispiel:

```
1 my @namen = ("Hans","Gabi","Moni");
2 sortiere (\@namen)
```

Der Subroutine `sortiere` wird durch Vorstellen eines Backslash vor den Variablennamen die Adresse der Liste `@namen` übergeben. Die Subroutine kennt nun die Adresse der Liste und kann über die Adresse die Werte der Liste lesen, die Werte sortieren und sortiert wieder unter der übergebenen Adresse abspeichern.

Somit entscheidet die Art der Argumentübergabe alleine, was ein Unterprogramm mit einem Argument machen kann.

Exkurs: Übergabemechanismus in PERL (für den Spezialisten)

Der Standardübergabemechanismus der Programmiersprache PERL ist eine Mischung von „Call by Value“ und „Call by Reference“. Diese Mischung führt bei ungeübten Programmieren schnell zu Verwirrung und Programmierfehlern und letztendlich zu einer Vermeidung von selbst geschriebenen Subroutinen. In PERL erinnert die Aufrufart einer Subroutine an „Call by Value“, jedoch werden intern dem Unterprogramm nicht eine Kopie der Argumente übergeben, sondern so genannte Aliase der einzelnen Argumente. Aliase sind Verweise auf die Speicheradresse einer Variablen. Eine Subroutine kann somit lesend und schreibend auf seine Argumente zugreifen. Problematisch wird diese Mischform, wenn Listen übergeben werden, da dann in PERL die Liste linearisiert wird und die Adressen der einzelnen Array-Elemente übergeben werden. Das Unterprogramm weiß gar nicht mehr, dass das Argument mit dem es aufgerufen wurde, eine Liste war. Erst mit der Version 5 der Programmiersprache PERL wurde der Übergabemechanismus „Call by Reference“ eingeführt. Jetzt können Argumente mit ihrer Adresse übergeben werden. Das Unterprogramm bekommt die Information über den Variablentyp und über die Adresse der aufrufenden Argumente. Skalare im aufrufenden Programm bleiben Skalare im Unterprogramm und Listen bleiben im Unterprogramm Listen. Auf jedes Argument kann über die Adresse zugegriffen werden.

Damit die verschiedenen Argumentübergabemethoden in PERL nicht zur Verwirrung führen, ist folgende Arbeitsweise hilfreich:

- Die Argumente dürfen bei selbst geschriebenen Subroutinen nur mit „Call by Reference“ übergeben werden dürfen. Das hat den Vorteil, dass die Subroutine die genaue Information über die Datenstruktur aller Variablen erhält und alle Variablen lesbar und schreibbar sind.

- Bei Systemroutinen werden die Argumente mit „Call by Value“ übergeben.

Setzt man den Referenzoperator \ vor den Variablennamen, wird in PERL automatisch die Adresse einer Variablen angesprochen.

`$name`	Wert der Variable `$name`
`\$name`	Adresse der Variable `$name`

Um einer Subroutine Argumente nach der Aufrufart „Call by Reference“ zu übergeben, muss man vor die Variablennamen der Subroutinenargumente immer einen Backslash schreiben.

3.1 Der STACK Speicher beim Subroutinenaufruf

Damit zwischen Unterprogrammen und aufrufendem Programm Daten ausgetauscht werden können, wird bei PERL ein spezieller Speicher reserviert, auf den sowohl das Unterprogramm als auch das aufrufende Programm zugreifen kann. Dieser Speicherbereich heißt STACK. Der STACK ist wie eine Liste organisiert und bekommt in PERL den reservierten Listenvariablennamen `@_`. In die Liste `@_` werden beim Aufruf des Unterprogramms automatisch die Werte bzw. Adresse des Arguments bzw. der Argumentvariablen, je nach Aufrufart, eingetragen. Das Unterprogamm und das aufrufenden Programm kann aus dieser Liste die Werte lesen.

Beispiele: „Call by Value“, „Call by Reference“

Hier ein Beispiel zur Demonstration der Aufrufart „Call by Value” und „Call by Reference”:

```
1 #Hauptprogramm
2 {
3 my $text = "hallo";
4 my $name = "sepp";
5 my @farben = ("rot","gruen","blau");
6 }
```

Adressbuch des Hautprogramms:

NAME	ADRESSE	TYP und WERT
$text	1000	Typ: Skalar, Wert = "Hallo"
$name	2000	Typ: Skalar, Wert = "sepp"
@farben	3000	Typ: Liste, Werte = "rot"
	3020	"gruen"
	3040	"blau"

1. Fall: Aufruf einer Systemroutine $\Longrightarrow$ Aufrufsart = Call by Value. Die Argumente werden entweder als Konstante oder als Variable übergeben:

Aufruf:

```
print ("Hallo");
```

NAME	ADRESSE	TYP und WERT
STACKSPEICHER	10000	Typ: Skalar, Wert="hallo"

```
get ($name);
```

NAME	ADRESSE	TYP und WERT
STACKSPEICHER	10000	Typ: Skalar, Wert="sepp", Alias auf 2000

```
print(@farben);
```

NAME	ADRESSE	TYP und WERT
STACKSPEICHER	10000	Typ: Skalar, Wert="rot", Alias auf 3000
		Typ: Skalar, Wert="gruen", Alias auf 3020
		Typ: Skalar, Wert="blau", Alias auf 3040

```
sort(@farben);
```

NAME	ADRESSE	TYP und WERT
STACKSPEICHER	10000	Typ: Skalar, Wert="rot", Alias auf 3000
		Typ: Skalar, Wert="gruen", Alias auf 3020
		Typ: Skalar, Wert="blau", Alias auf 3040

2. Fall: Aufruf einer eigenen Subroutine $\Longrightarrow$ Aufrufsart = Call by Reference. Es muss die Adresse der Argumente übergeben werden. Durch Voranstellen eines Backslash wird automatisch die Adresse einer Variable angesprochen.

Aufruf:

```
aendere_farben(\@farben);
```

NAME	ADRESSE	TYP und WERT
STACKSPEICHER	10000	Typ: Liste, Alias auf 3000

```
konvertiere(\$name,\$text);
```

NAME	ADRESSE	TYP und WERT
STACKSPEICHER	10000	Typ: Skalar, Wert= 1000
	10010	Typ: Skalar, Wert=2000

3.2 Parameter von Subroutinen

Der Übergabemechanismus in PERL ist ziemlich einfach realisiert. Wie im letzten Kapitel erklärt, speichert das aufrufende Programm die Argumente entsprechend der Schreibweise der Argumente auf dem STACK. Das Unterprogramm muss wiederum die Argumente vom STACK lesen. Das Lesen der Argumente vom Stack wird in PERL nicht automatisch beim Subroutinenaufruf initiiert, sondern muss mithilfe von Zuweisungen selbst programmiert werden. PERL überprüft niemals, ob die Datentypen oder die Anzahl der Argumente und Parameter zwischen Aufruf und Unterprogrammdefinition übereinstimmen.

Hier unterscheidet sich PERL grundsätzlich von anderen höheren Programmiersprachen, wie PASCAL, JAVA, C++ etc., wo jeder falsche Aufruf eines Unterprogramms zu einer Fehlermeldung des Compilers führt. In PERL führt ein falscher Aufruf einer Subroutine im glücklichsten Fall zum Programmabbruch; im Allgemeinen rechnet das Programm aber falsch weiter. Vielleicht wird dieser Umstand in neueren Versionen von PERL korrigiert (Stand PERL 5). Ein kleiner Überprüfungsmechanismus wird in PERL dadurch eingeschaltet, dass bei Subroutinen im Subroutinenkopf ein Prototyp definiert wird. Bei einem Subroutinenkopf (mit Prototyp-Definitition) werden innerhalb der runden Klammern Anzahl und Typen der Argumente festgelegt.

Exkurs: Prototypen

Prototypen sind eigentlich Kontextschablonen in PERL und führen eine rudimentäre Typprüfung ein. Weiterhin kann man mit Prototypen deklarierte Subroutinen wie *built-in* Subroutinen aufrufen, und man erspart sich darüber nachzudenken, ob an der Stelle, an der aufgerufen wird, eine Referenz übergeben werden soll oder nicht.

Grundsätzlich werden Prototypen zwischen dem Funktionsnamen und den geschweiften Klammern angegeben. Für jedes Argument wird das entsprechende Zeichen (siehe unten) eingefügt. Will man eine Referenz übergeben, so stellt man dem Zeichen ein \ vor, erst dann tritt die Typprüfung in Kraft. Soll eine Referenz auf verschiedene Typen übergeben werden, kann man diese in `[ ]` stellen (siehe Beispiele unten). Die Zeichen der jeweiligen Typen und ihre Besonderheiten (wenn ohne \ verwendet) sind:

- `$` für Skalar, erzwingt skalaren Kontext (wenn mit einer Liste aufgerufen, wird deren Länge Übergeben)
- `@` für Array, „frisst“ alle weiteren Argumente
- `%` für Hash, „frisst“ alle weiteren Argumente
- `&` für Subroutine, es wird eine anonyme Subroutine erwartet (siehe unten)
- `*` für Glob (alles andere, v.a. `FILEHANDLES`)

Beim Aufruf gibt es folgende Besonderheit: Wenn die Subroutine als Referenz übergeben oder mit `&` aufgerufen wird, werden *keine* Prototypen verwendet.

```
1 sub my_sub ($){
2    <Programmcode>
3 }
```

Damit die Argumente, die im STACK abgelegt sind, im Unterprogramm als Variable zur Verfügung stehen, müssen die Argumentwerte lokalen Variablen des Unterprogramms zugewiesen werden. Der STACK ist in PERL als Liste organisiert. Die STACK-Variable hat den Namen `@_` . Die Parameter bekommen den Wert der Argumente aus dem STACK durch Zuweisung über die STACK-Variable `@_`:

```
<subroutine parameter section> ::=
                          my ( <reference variable>{,<reference
                             variable>} ) = @_;
```

Beispiel 1:

Aufruf im Hautprogramm:

```
suche_buchstabe_in_liste (\$aktueller_Buchstabe,
        \@Liste_alter_Buchstaben, \$Anzahl_Elemente_in_Liste);
```

Wertzuweisung in der Subroutine:

```
my ($Buchstabe,$Liste,$Anzahl) = @_ ;
```

Beispiel 2:

Ein Programmextrakt:

```
{
# Hautprogramm:
my $aktueller_Buchstabe;
my $buchstabe_in_liste;
my @Liste_alter_Buchstaben;
my $Anzahl_Elemente_in_Liste;
my $gefunden;

...
# Aufruf der Subroutine:
  suche_buchstabe_in_liste (\$aktueller_Buchstabe,
        \@Liste_alter_Buchstaben, \$Anzahl_Elemente_in_Liste);
...
}
```

Adressbuch des Hauptprogramms:

NAME	ADRESSE	TYP und WERT
`$aktueller_Buchstabe`	1000	Typ: Skalar, Wert= `"a"`

`@liste_alter_Buchstaben`	2000	Typ: Liste, Werte= `"b"`
		`"c"`
		`"d"`

`$Anzahl_Elemente_in_Liste`	3000	Typ: Skalar, Wert= 3

Aufruf der Subroutine `suche_buchstabe` mit den Adressen der Argumente (Call by Reference):
`suche_buchstabe_in_liste(\$aktueller_Buchstabe,`
`\@Liste_alter_Buchstaben, \$Anzahl_Elemente_in_Liste);`

ergibt folgenden Inhalt des STACK-Speichers:

```
aendere_farben(\@farben);
```

NAME	ADRESSE	TYP und WERT
STACKSPEICHER Variablenname `@_`	10000	Typ: Skalar, Wert = 1000 Typ: Liste, Wert = 2000 Typ: Liste, Wert = 3000

Prototyp einer Subroutine:

```
sub suche_buchstabe_in_liste(\$\@\$)
```

lesen der Werte aus dem STACK-Speicher:

```
my ($Buchstabe,$Liste,$Anzahl)= @_
```

ergibt:

Adressbuch innerhalb der Subroutine `suche_buchstabe_in_liste`:

NAME	ADRESSE	TYP und WERT
`$Buchstabe`	20000	Typ: Skalar, Wert = 1000
`$Liste`	20010	Typ: Liste, Wert = 2000
`$Anzahl`	20020	Typ: Skalar, Wert = 3000

3.3 Dereferenzieren von Parametern

Innerhalb einer Subroutine stehen bei der Übergabeart „Call by Reference“ als Parameter die Adressen der Argumentvariablen zur Verfügung. Möchte man den Wert einer Argumentvariablen lesen oder verändern, muss man bei jedem Zugriff zuerst die Adresse auflösen, um an den Wert der Variable zu gelangen. Diese Adressauflösung nennt man in der Informatik „Dereferenzierung“. Die Parameter müssen also immer dereferenziert werden. Eine Variable wird in PERL dereferenziert, indem die Variable in geschweifte Klammern eingeschlossen und entsprechend dem Typ des Parameters ein `@` oder `$` vorangestellt wird.

Beispiel:

Adressbuch des Hautprogramms:

NAME	ADRESSE	TYP und WERT
`$Anzahl_Elemente_in_Liste`	3000	Typ: Skalar, Wert = 3
`@Liste_alter_Buchstaben`	2000	Typ: Liste, Werte = `"a"`
		`"b"`
		`"c"`
		`"d"`

Aufruf:

```
1  suche_buchstabe_in_liste(\$aktueller_Buchstabe,
2          \@Liste_alter_Buchstaben,\$Anzahl_Elemente_in_Liste);
```

Lesen der Adresswerte der Argumente aus dem Stack:

```
my ($Buchstabe,$Liste,$Anzahl)= @_
```

ergibt:

Adressbuch innerhalb der Subroutine `suche_buchstabe_in_liste`

NAME	ADRESSE	TYP und WERT
`$Anzahl`	20020	Typ: Skalar, Wert = 3000
`$Liste`	20020	Typ: Liste, Wert = 2000

Die Variable `$Anzahl` hat innerhalb der Subroutine den Wert 3000. Die Subroutine weiß nun unter welcher Adresse es den Wert des Arguments finden kann. Es muss also bei jedem Zugriff auf die Argumentvariable zuerst die Adresse auflösen: Es muss dereferenziert werden:

```
print ( ${$Anzahl} );
```

oder : `${$Anzahl} = ${$Anzahl} + 1;`

oder : `print ( @{$Liste} );`

3.4 Rückgabewerte von Subroutinen

Ähnlich wie Funktionen in der Mathematik können Subroutinen in PERL einen Wert an das aufrufende Programm über ihren Namen zurückgeben. Die Subroutine muss dann innerhalb eines Ausdruck aufgerufen werden.

Z.B. in der Mathematik die Berechnung eines Sinus-Wertes: Die Funktion `sinus` berechnet den Sinus-Wert seines Arguments und gibt das Ergebnis über seinen Namen an den Aufrufer zurück:

$$y = sin(2.3)$$

oder

$$y = sin(2.3) * cos(4.0)$$

Welcher Wert zurückgegeben wird, legt der Programmierer mit dem `return`-Statement im Unterprogramm fest. Stößt die Abarbeitung einer Subroutine auf das `return`-Statement, dann wird das Unterprogramm verlassen und der Wert des Ausdrucks, der hinter dem `return` steht, als Ergebnis zurückgegeben. In PERL können die Werte mehrerer Ausdrücke zurückgegeben werden.

Beispiel:

```
1 return "hallo"; # Rueckgabe des Strings "hallo"
2 return $erg;    # Rueckgabe des Wertes in der Variable $erg
3 return @namen;  # Rueckgabe der Liste @namen
```

4

Lokale und globale Variablen

4.1 Lokale Variablen

Lokale Variablen werden häufig auch als lexikalische Variablen bezeichnet. Sie entsprechen den Ideen modularer Programmierung, d.h. man kann in jedem abgeschlossenen Teil des Programms sicher sein, dass kein Wert unerwünschterweise von außerhalb dieses Teils geändert werden kann. Dies erleichtert sehr die Fehlersuche und ist im übrigen unerlässlich, falls mehrere Teile eines größeren Programms von verschiedenen Personen geschrieben werden sollen.

Die Deklaration von lokalen Variablen geschieht mit `my`. Dabei können auch mehrere Variablen auf einmal deklariert werden, außerdem können sie auch gleich initialisiert werden:

```
1 my ($var1, $var2);
2 my @array = (2,5,7);
```

Im Gegensatz zu anderen Programmiersprachen (wie z.B. C) kann die Deklaration an jeder beliebigen Stelle des Programms stattfinden. Es gibt bei PERL keine strikte Trennung von einem Deklarationsteil und einem Anweisungsteil.

Gültigkeitsbereich von lokalen Variablen

Durch die Deklaration mit `my` wird der Gültigkeitsbereich der Variablen eingeschränkt auf den Block, innerhalb dessen sie deklariert worden ist. Nur innerhalb dieses Blockes kann schreibend oder lesend auf sie zugegriffen werden, außerhalb dieses Blockes ist sie *unsichtbar*. Sobald der Block, in dem die lokale Variable gültig ist, fertig durchlaufen worden ist, verliert die lokale Variable jeweils auch den ihr zugeordneten Wert.
(**Ausnahme:** *Reference Counting*)

Ein „Block“ in diesem Sinne ist

- alles was von geschweiften Klammern umschlossen ist (bei Verschachtelungen von geschweiften Klammern: das, was von den innersten geschweiften Klammern umschlossen ist);
- die komplette Datei, falls die Deklaration überhaupt nicht von geschweiften Klammern umschlossen ist;
- an `eval` übergebene Strings (Sonderfall, den wir hier nicht näher besprechen werden)

Nicht gültig sind lokale Variablen in von einem Block aus aufgerufenen Subroutinen, die in einem getrennten Block deklariert werden.

ABER: lexikalische Variablen sind gültig in allen auf derselben Verschachtelungsebene explizit deklarierten Subroutinen, denn die befinden sich ja noch im selben Block. Daran denkt man oft nicht automatisch, wenn man ein Hauptprogramm mit Subroutinen schreibt.

Beispiel:

```
1  #!/usr/bin/perl -w
2
3  use strict;
4
5  my %haeufigkeit = ( wort1 => 5, wort2 => 3, wort3 => 7);
6  ausgabe(%haeufigkeit);
7
8  # Ende main, Anfang Subroutinen:
9
10 sub ausgabe {
11  my %hash = @_;
12  foreach my $wort (sort keys%hash) {
13    print "$wort\t$hash{$wort}\n";
14  }
15 }
```

In diesem Beispiel ist die Hash-Variable `%haeufigkeit`, obwohl mit `my` deklariert, in der gesamten Datei gültig, weil die Deklaration auf keiner Ebene von geschweiften Klammern umschlossen wird. Auf dieser selben Verschachtelungsebene (Verschachtelungsebene Null sozusagen) wird hier die Subroutine deklariert – und dort ist dann `%haeufigkeit` noch gültig!

Im Sinne einer modularen Programmierung ist das natürlich unerwünscht. Gelöst werden kann dieses Problem dadurch, dass das Hauptprogramm in geschweifte Klammern eingeschlossen wird, um den Gültigkeitsbereich der Hauptprogramm-Variablen auch tatsächlich aufs Hauptprogramm zu beschränken.

Die Schleifenvariable in einer `for`/`foreach`-Anweisung ist innerhalb des Blockes gültig, der von den geschweiften Klammern der Schleife umschlossen wird . Sie wird intern so behandelt, als wäre sie innerhalb dieser Klammern deklariert worden.

4.2 Reference Counting

Definition (Reference Counting):
Weiterexistenz der Inhalte von lokalen Variablen außerhalb ihres Gültigkeitsbereichs.

Reference Counting bedeutet, dass die Inhalte, auf die durch Referenzen verwiesen wird, auch nach dem Verlassen ihres lexikalischen Gültigkeitsbereichs weiterexistieren, wenn irgendetwas *länger Lebendes* (z.B. eine globale Variable, eine weiter existierende Subroutine) noch die Referenzen verwendet.

```
1  #!/usr/bin/perl -w
2
3  @global_array = ();
4  {
5    save_as_array(2,4,6);
6    save_as_array(3,5,7);
7    foreach my $array_ref (@global_array) {
8      foreach my $elem (@$array_ref) {
9        print "$elem ";
10     }
11     print "\n";
12   }
13 }
14
15 sub save_as_array {
16  my @arguments = @_;                    # receive array-references
17  push (@global_array, \@arguments);     # store them
18 }
```

Dieses Programm funktioniert, obwohl der Gültigkeitsbereich von `@arguments` auf `save_as_array` beschränkt ist. Der Inhalt bleibt im Speicher erhalten, da die globale Variable `@global_array` noch die Referenz enthält.

Reference Counting kann man auch gezielt zur Schaffung von lokalen Variablen benutzen, die nur in einer oder mehreren ganz bestimmten Subroutinen gültig sind, wobei sie über die gesamte Dauer des Programms hinweg ihren Wert behalten – eine Eigenschaft, die normalerweise nur globale Variablen haben.

```
1  #!/usr/bin/perl -w
2
3  {
4  print_counter();
5  increase_counter();
6  print_counter();
7  decrease_counter();
8  print_counter();
9  }
10
11
12 BEGIN {
13   my $counter = 42;
14   sub increase_counter {$counter++}
15   sub decrease_counter {$counter--}
16   sub print_counter {
17       print "Counter = $counter\n";
18   }
19 }
```

Normalerweise würde `$counter` initialisiert werden, und dann seinen Wert nach Verlassen des Blockes wieder verlieren. Da aber die Subroutinen `$counter` verwenden (und diese Subroutinen ja nicht aufhören zu existieren), bleibt auch der Wert von `$counter` über die gesamte Programmlaufzeit erhalten. Die Anweisung `BEGIN` sorgt dafür, dass der Block, in dem `$counter` initialisiert wird, als erster Programmteil abgearbeitet wird – so kann man vermeiden, dass die Subroutinen aufgerufen werden ohne dass `$counter` initialisiert worden wäre.

4.3 Globale Variablen

Globale Variablen sind solche Variablen, auf die von jeder beliebigen Stelle im gesamten Programm aus (d.h. auch von allen Subroutinen aus) sowohl lesend als auch schreibend zugegriffen werden kann. Globale Variablen sollten wegen dieser Eigenschaft im Allgemeinen möglichst vermieden werden – man weiß nie, in welchem anderen Programmteil der Wert der Variablen nicht eventuell geändert wird, sei es bewusst oder unbewusst (weil z.B. innerhalb eines größeren Programms an anderer Stelle eine Variable gleichen Namens noch einmal definiert wird). Dennoch sind globale Variablen manchmal sinnvoll, z.B. wenn sie wirklich fast überall im Programm benötigt werden und andernfalls in praktisch jedem Subroutinen-Aufruf explizit als Argument übergeben werden müssten.

Per default sind alle in einem PERL-Programm verwendeten Variablen globale Variablen, d.h. man muss sie nicht irgendwie besonders deklarieren, damit sie global gültig sind. Ab PERL-Version 5.6. können sie mit `our` deklariert werden (z.B. `our $var;`), müssen aber nicht. Die Deklaration mit `our` erhöht zwar einerseits die Lesbarkeit des Programms, da auf diese Weise globale Variablen klar als solche gekennzeichnet werden. Andererseits aber sind diese Programme dann unter älteren PERL-Versionen nicht lauffähig.

Wozu globale Variablen?

Da globale Variablen an jeder Stelle des Programms veränderbar sind, erhöhen sie die Anfälligkeit eines Programms für unbeabsichtigte Verwendung zweier Variablen mit demselben Namen und den Aufwand bei der Fehlersuche, da man nicht von vornherein wissen kann, wo diese Variable verändert worden sein könnte.

Dennoch gibt es ein paar Fälle, für die globale Variablen empfehlenswert sind:

(a) Die Variable wird in sehr vielen unterschiedlichen Blöcken des Programms gebraucht, und es wäre sehr lästig, sie jedesmal als Argument weiterzureichen.

(b) Man möchte die Möglichkeit haben, den Befehl `local` auszunutzen (werden wir noch besprechen).

(c) Die Variable soll von Programmcode benutzt werden, der nicht in derselben Datei steht (wie das insbesondere bei Modulen der Fall ist, die wir noch behandeln werden).

Punkt (a) kann man in manchen Fällen (wie gesehen) mittels Reference Counting lösen; sollte auch dies nicht mehr praktikabel sein, weil zu viele Subroutinen die Variable nutzen sollen, kann man zur Not außerhalb aller Blöcke eine lexikalische Variable (mit `my`) deklarieren, die damit automatisch in der ganzen Datei gilt. Dafür gibt es allerdings eine flexiblere Lösung in PERL.

Die Ziele (b) und (c) hingegen sind mit lokalen Variablen prinzipiell nicht zu erreichen.

Die Verwendung von `our`

Wird kein `use strict` verwendet, ist automatisch jede Variable, die nicht speziell mit `my` deklariert wird, eine globale Variable. Wird jedoch (empfehlenswerterweise) `use strict` verwendet, können globale Variablen nur mit `our` deklariert werden.

Wie bereits erwähnt wurde, sind globale Variablen von überall aus ansprechbar. Es gibt jedoch prinzipiell zwei Arten, globale Variablen anzusprechen: mit und ohne Package-Namen. (Packages werden später genauer vorgestellt.) Die Möglichkeit, die globale Variable ohne explizite Nennung ihres Package-Namens anzusprechen, gilt nur in denjenigen Blöcken, in denen sie mit `our` deklariert wurde. Ihr Wert bleibt aber stets erhalten, und über den Package-Namen ist sie auch von anderen Stellen des Programms her jederzeit ansprechbar.

Beispiel:

```
 1  #!/usr/bin/perl
 2
 3  use strict;
 4  use warnings;
 5
 6  {
 7    our @liste = (2,3,4);
 8    bla1();
 9    bla2();
10    bla3();
11  }
12
13  sub bla1 {
14      our @liste;
15      print "Der Inhalt der Liste ist ", @liste, "\n\n";
16  }
17
18  sub bla2 {
19      print "Der Inhalt der Liste ist ", @main::liste, "\n\n";
20  }
```

```
21
22 sub bla3 {
23     print "Der Inhalt der Liste ist ", @liste, "\n\n";
24 }
```

Im Hauptprogramm (das in einen eigenen Block gefasst ist), wird `@liste` deklariert und mit einem Wert belegt. In `bla1` wird `@liste` dann noch mal deklariert, der im Hauptprogramm zugewiesene Wert ist somit abrufbar (und auch veränderbar). In `bla2` wird `@liste` nicht noch mal deklariert. Hier ist die Variable nur über ihren Package-Namen ansprechbar. In `bla3` ist `@liste` nicht erneut deklariert worden, trotzdem wird versucht, sie ohne den Package-Namen anzusprechen. Dies führt dann zu folgender Fehlermeldung beim Aufruf des Programms:

```
Variable "@liste" is not imported at test.perl line 23.
Global symbol "@liste" requires explicit package name at test.perl line 23.
Execution of test.perl aborted due to compilation errors.
```

Verwendet man `use strict` und verzichtet man gleichzeitig auf das Ansprechen von globalen Variablen über ihren Package-Namen, so ergibt sich auch für globale Variablen die Möglichkeit zu einer recht klaren Verwendung:

- Soll die Variable in einem bestimmten Block angesprochen werden, muss sie auch in diesem Block deklariert worden sein. Dies lässt sofort erkennen, von welchen globalen Variablen das Funktionieren des Blockes abhängig ist. Das ist zum einen hilfreich bei der Fehlersuche, zum anderen macht es sofort klar, unter welchen Bedingungen ein Block läuft, falls man ihn in ein anderes Programm kopiert.
- Wird die Variable in einem Block angesprochen, in dem sie nicht deklariert worden ist, so erzeugt das Programm eine Fehlermeldung. Dies zwingt einen dazu, explizit zu planen, in welchen Blöcken die Variable genutzt werden soll und in welchen nicht.

4.4 Hauptunterschied zwischen lokalen und globalen Variablen

Lokale Variablen sind grundsätzlich nur in dem Block gültig und ansprechbar, in dem sie deklariert worden sind. In dem Moment, in dem die Programmabarbeitung das Ende des betreffenden Blocks erreicht, geht auch der Wert verloren, der in der lokalen Variablen gespeichert war – sofern er nicht explizit an eine andere Variable übergeben wird (z.B. mittels `return`) oder das Reference Counting greift.

Globale Variablen sind grundsätzlich von jeder Stelle eines Programms aus ansprechbar, bei Einbindung einer Datei auch von Programmteilen außerhalb der Datei. Die Variable verliert nicht ihren Wert, wenn die Programmabarbeitung an ein Blockende kommt oder ähnliches.

4.5 Konkurrenz von globalen und lexikalischen Variablen

Existiert in einem bestimmten Block nach entsprechenden Deklarationen sowohl eine lexikalische als auch eine globale Variable desselben Namens, dann wird die zuletzt deklarierte Variable angesprochen.

Beispiel:

```
1 #!/usr/bin/perl
2
3 use strict;
4 use warnings;
5
6 our $var = 7;
7
```

```
 8 {
 9   print "$var\n"; # die globale Variable wird ausgegeben
10   my $var = 10;
11   print "$var\n"; # die lokale Variable wird ausgegeben
12 }
```

Das Programm gibt beim ersten Mal den Wert 7 aus, beim zweiten Mal den Wert 10.

4.6 Der Befehl `local`

Zur vorübergehenden Benutzung von globalen Variablen in Subroutinen, die nach Abarbeiten der Subroutine automatisch wieder ihren vorherigen Wert annehmen sollen. Dies ist besonders empfehlenswert für globale Variablen, die von PERL vordefiniert sind, wie z.B. `$_` oder `$/`.

Werden solche globalen Variablen wie z.B. `$_` oder `$/`, die in PERL eine besondere Funktion haben, in einer Subroutine verändert, so bleibt die Veränderung auch nach Rückkehr aus der Subroutine gültig – es sind ja globale Variablen. Dies ist jedoch sehr gefährlich, da man bei Verwendung dieser Variablen normalerweise von der *Default*-Belegung ausgeht und sich dann wundert, dass das Programm ganz anders abläuft als erwartet.

Beispiel: Manipulation von `$/`.

`$/` ist der Input Record Separator. Diese Variable legt fest, bis wohin beim Lesen von einem Filehandle jeweils eingelesen wird. *Per Default* ist die Variable mit `\n` belegt, d.h. es wird immer bis zum nächsten *newline* eingelesen, z.B. in

```
1 while (<FILE>) {
2
3     print "The next line in the file is $_\n";
4
5 }
```

Diese Variable kann aber auch beliebig neu belegt werden. Soll z.B. bis zum nächsten Punkt gelesen werden, statt bis zum nächsten *newline*, genügt ein einfaches `$/ = ".";`

Dies wird in folgendem Beispiel gemacht. Wird der Wert von `$/` allerdings nicht zurückgesetzt, so gilt er weiter für alle Aufrufe von File-Handles, inklusive des Einlesens von `STDIN` mit `<>`. Auch dort wird dann jeweils nur bis zum Punkt eingelesen, und daher produziert das folgende Programm etwas unerwartete Resultate:

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5
 6 {
 7     my $file = "sentences.txt";
 8     print "Reading sentences from file $file\n";
 9     read_sentences_from_file($file);
10
11     print "Please enter the file name again: ";
12     my $file2 = <>;
13     print "You entered: $file2\n";
14 }
15
16 sub read_sentences_from_file {
17
18     my $filename = shift;
19
```

```
20      open IN, $filename or die "$!";
21      $/ = ".";
22      while (<IN>) {
23          print "$_\n";
24      }
25      close IN or die "$!";
26  }
```

Das Problem kann gelöst werden, indem am Ende der Subroutine `$/` mit `$/="\n"` die Variable wieder auf den *Default*-Wert gesetzt wird. Aber auch das kann unerwünscht sein, wenn die Variable in der aufrufenden Routine schon auf einen anderen Wert gesetzt war. Deshalb ist es die beste Lösung, die Variable zu Beginn der Subroutine mit `local($/);` zu lokalisieren. Sie kann dann in der Subroutine nach Belieben manipuliert werden, bei Rückkehr in die aufrufende Routine nimmt sie automatisch wieder den Wert an, den sie vorher hatte. Nur diese Lokalisierung garantiert eine problemlose Wiederverwendung der Subroutine in anderen Programmen. Es sollte immer darauf geachtet werden, dass eine Subroutine globale Variablen unverändert an das Hauptprogramm zurückgibt, es sei denn, es ist die offensichtliche und explizit erwünschte Funktion der Subroutine, diese globale Variable verändert zurückzugeben. Gerade bei den PERL-Systemvariablen dürfte das allerdings praktisch nie der Fall sein, dass deren Wert explizit verändert werden soll, und gerade diese Variablen sollten immer mit ihrer ursprünglichen Belegung zurückgegeben werden.

Der in der Praxis häufigste Fall, bei dem diese Problematik zu beachten ist, dürfte die Verwendung von `$_` sein.

`local` vs. `my` in Bezug auf verschachtelte Subroutinen

Lokale Variablen sind nur in dem Block gültig, in dem sie deklariert werden – in Subroutinen, die aus diesem Block heraus aufgerufen werden, sind sie hingegen nicht sichtbar. Man kann ihren Gültigkeitsbereich also direkt aus dem Programmcode her ersehen: vom Beginn des Blocks bis zu seinem Ende können sie angesprochen werden, in anderen Teilen des Programmcodes nicht. Daher spricht man auch von lexikalischen Variablen, bzw. auf Englisch von *lexically scoped variables*. (*textually scoped* wäre vielleicht der etwas klarere Begriff gewesen, aber *lexically scoped* hat sich halt eingebürgert).

Globale Variablen hingegen sind grundsätzlich von überall im Programmcode her ansprechbar. Wird ihr aktueller Wert mit `local` vorübergehend unsichtbar gemacht und durch einen anderen ersetzt, so gilt diese Änderung zwar bis zum Verlassen des Blockes, aber nicht wie bei `my` im lexikalischen Sinn: stattdessen gilt der temporäre Wert auch in allen zwischenzeitlich aufgerufenen Subroutinen, also in einem zeitlichen Sinne, von der `local`-Anweisung bis zum Verlassen des Blocks. Man spricht daher auch von *dynamischen Variablen* bzw. auf Englisch *dynamically scoped varibles*.

Beispiel 1:

```
1  $variable = "hello";
2  print "\$variable is $variable in the body.\n";
3  temporary();
4  print "\$variable is still $variable in the body.\n";
5
6  sub temporary {
7      my $variable = "goodbye";
8      print "\$variable is $variable in the temporary sub.\n";
9      inner();
10 }
11
12 sub inner {
13     print "\$variable is $variable in the inner sub.\n";
14 }
```

Beispiel 2:

```
1  $variable = "hello";
2  print "\$variable is $variable in the body.\n";
3  temporary();
4  print "\$variable is still $variable in the body.\n";
5
6  sub temporary {
7      local $variable = "goodbye";
8      print "\$variable is $variable in the temporary sub.\n";
9      inner();
10 }
11
12 sub inner {
13     print "\$variable is $variable in the inner sub.\n";
14 }
```

4.7 Packages

Packages sind abgetrennte „Namensräume“ für globale Variablen (in anderen Programmiersprachen unter *name spaces* bekannt). Diese Namensräume haben nichts mit den beschränkten Gültigkeitsbereichen lokaler Variablen zu tun, sondern dienen nur einer übersichtlichen Verwaltung der globalen Variablen. Sofern sie in verschiedenen Packages deklariert sind, kommen sich Variablen gleichen Namens nicht in die Quere.

Man kann von jedem Programmteil auch auf globale Variablen aus anderen Packages zugreifen, man muss sie nur entsprechend *qualifizieren*, d.h. den entsprechenden Paktenamen voranstellen. Das aktuelle Package in einem Programm ist defaultmäßig `main`.

Beispiel 1:

```
1  $var = 5;
2  print "$var\n";    # prints 5
3  print "$main::var\n";   # prints 5
```

Bei Verwendung eines Variablennamens ohne Package-Zusatz wird (abgesehen von 2 Ausnahmen) die Variable des aktuellen Packages mit diesem Namen angesprochen.

Beispiel 2:

```
1  #!/usr/bin/perl -w
2  #Programm packages
3
4  package Alpha;
5  $name = "rainer";
6
7  package Beta;
8  $name = "bernd";
9
10 package main;
11 print "name (= main::name) : $name\n";
12 print "Alpha::name : $Alpha::name\n";
13 print "Beta::name : $Beta::name \n";
```

Die eine Ausnahme von der Regel, dass bei unqualifizierten Variablennamen auf die globale Variable dieses Namens im selben Package zurückgegriffen wird, liegt vor, wenn sie von einer lexikalischen Variablen gleichen Namens überlagert wird.

Die andere Ausnahme sind PERLs Spezialvariablen wie z.B. `$_` und `$/`. Diese werden auch bei Verwendung in anderen Packages stets dem Package `main` zugeordnet.

Packages und lexikalische Variablen

Auf lexikalische Variablen hat es im Gegensatz zu globalen Variablen keinerlei Einfluss, innerhalb welchen Packages sie deklariert werden. Lexikalische Regeln *leben* in Speichern für lexikalische Variablen (normalerweise an Blöcke gebunden) und haben nichts mit Packages zu tun. Globale Variablen leben in Packages im Speicher für globale Regeln und haben nichts mit Blöcken zu tun.

Beispiel 3:

```
1  package Alpha;
2  my $a = 10;
3     $x = 5;
4
5  package Beta;
6  my $b = 20;
7     $x = 3;
8
9  package main;
10 print "$a, $b, $x, $Alpha::x, $Beta::x\n";
```

Was wird ausgegeben?

Lexikalische und globale Variablen gehorchen völlig unterschiedlichen Regeln und haben nichts miteinander zu tun.

Die Beispiele mit mehreren Packages in einem Programm dienen nur der Demonstration; normalerweise ist es eher unüblich, mehrere Packages in einer einzigen Datei zu deklarieren.

Packages und `our`

Im Zusammenhang mit Packages lässt sich auch `our` besser verstehen: eine Deklaration mit `our` erledigt zwei Dinge:

1. Sie erzeugt eine globale Variable im aktuellen Package.
2. Sie schreibt einen Verweis auf diese globale Variable in den Speicher für lexikalische Variablen. Dieser Verweis findet sich im Speicher für lexikalische Variablen unter dem Namen der globalen Variablen (ohne den Package-Namen), d.h. er wird genau dann gefunden, wenn eine lexikalische Variable dieses Namens gesucht wird. Wenn z.B. im Package `main` die Deklaration `our $var` erscheint, so wird `$main::var` angelegt, und im Speicher für lexikalische Variablen unter dem Namen `$var` ein Verweis auf `$main::var` angelegt. Wird der Verweis angesprochen, leitet er die Anfrage nach dem Wert sozusagen an die globale Variable weiter, auf die er verweist. Dadurch ist die globale Variable so ansprechbar, als wäre sie eine lexikalische Variable. Verändert wird aber stets nur der Wert der einen globalen Variablen, selbst wenn sie in verschiedenen Blöcken mit `our` redeklariert und von dort aus verändert wird.

Beispiel 4:

```
1  use warnings;
2  use strict;
3
4  my $var = 7;
5
6  print "\$var = $var\n";
7  print "\$main::var = $main::var\n";
```

Das Programm gibt nur für `$var` einen Wert aus, nicht für `$main::var`, weil `$var` nur im Speicher für lexikalische Variablen steht, nicht im Package `main`, das die globalen Variablen beinhaltet.

Beispiel 5:

```
1 use warnings;
2 use strict;
3
4 our $var = 7;
5
6 print "\$var = $var\n";
7 print "\$main::var = $main::var\n";
```

Das Programm gibt sowohl für `$var` als auch für `$main::var` einen Wert aus.

Die übliche Verwendung von `our` ist folgende:

- Man deklariert zu Programmbeginn außerhalb jeden Blocks alle globalen Variablen mit `our`, ggfs. mit einer Initialisierungszuweisung. Dies macht im Gegensatz zu `my` deutlich, dass es sich um globale Variablen handelt.
- Je nach Programmierstil führt man zu Beginn einer Subroutine (oder ggfs. anderen Blöcken) neben neu zu schaffenden lexikalischen Variablen die `our`-Deklaration erneut auf, jedoch ohne Zuweisung. Dieser Programmierstil hat 2 Vorteile:
  1. Es ist in jeder Subroutine sofort ersichtlich, welche globalen Variablen sie benutzt (besonders vor "copy & paste"-Aktionen nützlich);
  2. Sollte in einem umfassenderen Block außerdem noch eine echte lexikalische Variable gleichen Namens existieren, so wird sie ausgeblendet – es wird auf jeden Fall die globale Variable angesprochen.

4.8 Übungsaufgaben

Aufgabe 4.1 Schreibe zu folgendem Hauptprogramm entsprechende Subroutinen. Die Subroutinen sollen alle gemeinsam eine Variable `$summe` benutzen, die von anderen Teilen des Programms aus nicht verändert werden kann (außer über die Funktionsaufrufe). Zum Programmstart soll die Summe grundsätzlich 200 Euro betragen.

```
1 {
2   summe_anzeigen();
3   einzahlen(500);
4   auszahlen(300);
5   summe_anzeigen();
6   einzahlen(150);
7   auszahlen(25);
8   summe_anzeigen();
9 }
```

Aufgabe 4.2 Schreibe eine Gegenüberstellung der Eigenschaften von `my` und `our` (Sichtbarkeitsbereich der Variablen, Ende der Existenz, Möglichkeit der Anwendung von `local`, etc.; nicht als Programm, sondern als Text)

Aufgabe 4.3 Kopiere das Programm `false_friends.perl` in ein Verzeichnis. Lass es zuerst so laufen wie es ist, korrigiere es, und lasse es dann noch mal laufen (die korrigierte Version des Programms bitte einreichen).

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

# liest Namen von Personen ein und zu jeder Person
# auch die Namen von Freunden dieser Person

use strict;

{
  my %freunde;           # Key: Name, Wert: Namen der Freunde
  my $freunde_aref;      # Ref. auf array mit Namen der Freunde
  print "Bitte einen Namen eingeben: ";
  while (<>) {
    chomp;
    last unless $_;
    $freunde_aref = freunde_einlesen($_);
    $freunde{$_}=$freunde_aref;
    print "Bitte einen Namen eingeben: ";
  }
  foreach my $name (sort keys%freunde) {
    print "$name: ", join(' ', @{$freunde{$name}}), "\n";
  }
}

sub freunde_einlesen {
  my $name = shift;
  my $aref;       # Ref. auf array mit Namen der Freunde
  print "Bitte eine(n) Freund(in) von $name eingeben: ";
  while (<>) {
    chomp;
    last unless $_;
    push (@$aref, $_);
    print "Bitte eine(n) Freund(in) von $name eingeben: ";
  }
  return $aref;
}
```

Aufgabe 4.4 Welche Werte gibt diese Version des Programms aus, und warum?

```perl
our $var = 7;

{
   print_var();
}

sub print_var {
   my $var = 10;
   print "\$var = $var\n";
   print "\$main::var = $main::var\n";
}
```

Aufgabe 4.5

(a) Was liefert Beispiel 3 in Abschnitt 4.7 als Ergebnis, wenn man `$x` beide Male mit `our` deklariert?

(b) Was genau geschieht intern beim Abarbeiten der ersten und der zweiten Zeile mit `our $x = ...`? Bitte beantworte die folgenden Fragen:

- Wie heißt die Variable genau (d.h. inklusive Package-Namen), deren Wert beim ersten `our $x = ...` belegt wird?
- Was passiert dabei im Speicher für lexikalische Variablen?
- Wie heißt die Variable genau (d.h. inklusive Package-Namen), deren Wert beim zweiten `our $x = ...` belegt wird?
- Was passiert dabei im Speicher für lexikalische Variablen?
- In welchem Speicher wird bei `print "..., $x, .."` zuerst ein Wert für die Variable gesucht?
- Was wird dort gefunden?
- Wie heißt die Variable genau, die letztlich angesprochen wird?

5

Sortieren von Listen

5.1 Sortieren von Listen mit dem `sort`-Befehl

Vorbemerkung: `qw`

Wenn man Strings in einer Liste schreibt, muss man normalerweise alle Strings einzeln mit Anführungszeichen versehen und mit Kommas trennen. Diese Arbeit kann man sich sparen, wenn man `qw` vor die Liste setzt:

("das", "sind", "meine", "woerter")

ist also äquivalent zu

qw(das sind meine woerter)

Aufpassen muss man allerdings, wenn man Strings in die Liste schreiben will, die Leerzeichen enthalten. Dann funktioniert `qw` nicht wie gewünscht, da es Leerzeichen als Trenner zwischen Strings interpretiert.

qw(frankfurt berlin new york)

ist also leider **nicht** äquivalent zu

("frankfurt", "berlin", "new york")

sondern zu

("frankfurt", "berlin", "new", "york").

sort `@liste`

Die Überschrift dieses Abschnitts zeigt schon den Standard-Aufruf von `sort` in PERL. Die Wirkung ist nicht etwa, dass danach der Inhalt von `@liste` sortiert ist, sondern nur der Rückgabewert des Ausdrucks `sort @liste` ist der sortierte Listeninhalt. Diesen Rückgabewert kann man abfragen oder ausdrucken – der Inhalt von `@liste` selbst bleibt aber unverändert.

Beispiel:

```
1 @liste = qw(a c b);
2 sort @liste;
3 print join(", " @liste);
```

Ausgabe: a, c, b

Und vorher noch die Warnmeldung:

```
Useless use of sort in void context at test.perl line 7.
```

Also, wenn schon, dann

```
1 @liste = qw(a c b);
2 print join(", " sort @liste);
```

Danach ist der Inhalt von `@liste` immer noch unverändert, aber ausgegeben hat man die sortierte Liste.

Wie wird sortiert?

Was gibt wohl das folgende Programm aus?

```
1 #!/usr/bin/perl -w
2
3 use strict;
4
5 my @liste = (12, 10, 100, 11);
6 print join(", ", sort @liste), "\n";
```

Wie man bei Ausführen des Programms merkt, sortiert der `sort`-Befehl *per default* alphabetisch, nicht numerisch – und deshalb kommt in der Ausgabe z.B. 100 vor 11 (wie beim alphabetischen Sortieren auch "baa" vor "bb" kommen würde).

Eigene Sortierungen festlegen

Was muss man also tun, wenn man nicht alphabetisch sortieren will, sondern z.B. numerisch? Dann muss in einer eigenen Subroutine festlegen wie sortiert werden soll. Diese Subroutine muss eine Antwort auf folgende Frage liefern: Wenn ich zwei Elemente der zu sortierenden Liste habe, welches der beiden soll dann vor dem anderen einsortiert werden? Von PERL aus werden der Subroutine die zwei zu vergleichenden Listenwerte als Inhalte der Variablen `$a` und `$b` zur Verfügung gestellt (diese Variablen braucht man also nicht selbst zu deklarieren). Der Rückgabewert der Subroutine wird in folgender Weise als Antwort interpretiert:

```
Rückgabewert < 0: $a soll vor $b einsortiert werden

Rückgabewert == 0: egal

Rückgabewert > 0: $a soll hinter $b einsortiert werden
```

Beispiel:

```
 1 my @liste = (12, 10, 100, 11);
 2 print join(", ", sort num_sort @liste), "\n";
 3
 4 sub num_sort {
 5
 6   if ($a < $b) {
 7     return -1;  # $a vor $b einsortieren
 8   } elsif ($a == $b) {
 9     return 0;  # Sortierreihenfolge egal
10   } elsif ($a > $b) {
11     return 1;  # $a hinter $b einsortiern
12   }
13
14 }
```

Ausgabe: 10, 11, 12, 100

Es gibt auch einen Vergleichsoperator in PERL, der genau denselben Rückgabewert liefert wie unsere Subroutine `num_sort`, nämlich <=>. `$x <=> $y` liefert also eine -1, wenn `$x` kleiner ist als `$y`, eine 1, wenn `$x` größer ist als `$y`, und eine Null, wenn beide Werte gleich groß sind. Wir hätten die Subroutine also auch viel kürzer schreiben können:

```
1  sub num_sort {
2
3    return $a <=> $b;
4
5  }
```

Und es geht noch kürzer. Man kann nämlich den Block der Subroutine auch direkt hinter den `sort`-Befehl schreiben, ohne der Subroutine überhaupt einen Namen zu geben:

```
1  my @liste = (12, 10, 100, 11);
2  print join(", ", sort {$a <=> $b} @liste), "\n";
```

Auch diese Version gibt eine numerisch sortierte Liste aus.

Insgesamt gibt es also 3 Arten, den sort-Befehl aufzurufen:

1. sort LISTE
2. sort SUBROUTINENAUFRUF LISTE
3. sort BLOCK LISTE

Der `cmp`-Vergleichsoperator

Der `cmp`-Vergleichsoperator macht dasselbe wie der <=>-Operator, nur dass der Vergleich alphabetisch erfolgt statt numerisch.

```
1  print join(", ", sort {$a cmp $b} qw(einige wenige dinge)), "\n";
```

würde also folgende **Ausgabe** erzeugen: `dinge, einige, wenige`

Das ist gleichzeitig die *default*-Sortierung, d.h. das entspricht genau dem Aufruf von `sort` ohne eigene Subroutine.

Absteigend sortieren

Um absteigend zu sortieren, muss nur dafür sorgen, dass die Rückgabewerte der Subroutine bzw. des Blocks der gewünschten Einsortierung entsprechen. In unserem ersten Beispielprogramm für numerische Sortierung brauchen wir dazu lediglich das Größer- und das Kleinerzeichen in den `if`-Bedingungen zu vertauschen:

```
1  my @liste = (12, 10, 100, 11);
2  print join(", ", sort num_sort_reverse @liste), "\n";
3
4  sub num_sort_reverse {
5    if ($a > $b) {
6      return -1;  # $a vor $b einsortieren
7    } elsif ($a == $b) {
8      return 0;  # Sortierreihenfolge egal
9    } elsif ($a < $b) {
10     return 1;  # $a hinter $b einsortiern
11   }
12 }
```

Jetzt wird beim Vergleich von 2 Elementen das größere vor dem kleineren einsortiert, d.h. es wird absteigend sortiert. In der Subroutine kann man die Bedingungen wie folgt umformulieren, ohne ihre Aussage zu verändern:

```
sub num_sort_reverse {
  if ($b < $a) {
    return -1;  # $a vor $b einsortieren
  } elsif ($b == $a) {
    return 0;  # Sortierreihenfolge egal
  } elsif ($b > $a) {
    return 1;  # $a hinter $b einsortiern
  }
}
```

Das entspricht genau `$b <=> $a`, wie man leicht feststellen kann, wenn man die `if`-Bedingungen und die zugehörigen Rückgabewerte nacheinander betrachtet. D.h. man kann das ganze Programm auch so schreiben:

```
1 my @liste = (12, 10, 100, 11);
2 print join(", ", sort {$b <=> $a} @liste), "\n";
```

Die einfachste Art, sich die Funktionsweise von <=> zu merken, ist vermutlich, es als die gleichzeitige Abfrage auf kleiner, gleich oder größer zu betrachten und sich zu merken, dass immer folgendes gilt:

- ist „kleiner" wahr, wird `$a $b` sortiert.
- ist „größer"wahr, wird `$b $a` sortiert.

5.2 Sortieren nach mehrfachen Kriterien

Manchmal will man eine Liste nach einem Kriterium sortieren (z.B. Zahlen danach, ob sie gerade oder ungerade sind), und im Falle der Gleichheit bezüglich dieses Kriteriums noch nach einem weiteren Kriterium (z.B. nach ihrer Größe).

Um eine Liste von Zahlen so zu sortieren, dass zuerst alle geraden, dann alle ungeraden Zahlen ausgegeben werden, würde man zunächst einmal so etwas schreiben:

```
1 my @liste = (3, 2, 7, 8, 10, 5);
2 print join(", ", sort { ($a % 2) <=> ($b %2) } @liste), "\n";
```

Ist `$a` gerade und `$b` ungerade, so ergibt `$a % 2` den Wert 0 und `$b % 2` den Wert 1. Es ergibt sich also der "kleiner"-Fall. Somit wird `$a` vor `$b` einsortiert. Ist `$a` ungerade und `$b` gerade, ergibt sich 1 $>$ 0, `$a` wird hinter `$b` einsortiert. Die geraden Zahlen werden also immer vor die ungeraden sortiert.

Kurzer Exkurs: Die Modulo-Funktion

Die Modulo-Funktion ergibt den Rest einer Division. So ergibt z.B. `7 : 3` den Wert `2, Rest 1`. `7 modulo 3` würde daher `1` ergeben. In PERL wird die modulo-Funktion mit einem `%` ausgedrückt: `$rest = 7 % 3`.

Will man untersuchen, ob eine Zahl gerade ist oder ungerade, kann man dies mit einer Division durch 2 tun: bei geraden Zahlen ergibt sich kein Rest, bei ungeraden Zahlen ergibt sich immer der Rest 1:

```
1 if ($x % 2) {
2   print "Die Zahl ist ungerade.\n";
3 } else {
4   print "Die Zahl ist gerade.\n";
5 }
```

Jetzt wollen wir aber genau in den Fällen noch weitersortieren, in denen entweder beide Zahlen gerade oder beide Zahlen ungerade sind, d.h. wo der Vergleichsoperator eine Null zurückgibt. Will man etwas genau dann ausführen, wenn eine vorhergehende Anweisung zu dem Ergebnis Null geführt hat, dann bietet sich der `or`-Operator an, d.h. das logische Oder.

Exkurs 2: Das logische Oder

Die logische Oderverknüpfung ergibt folgende Werte:

```
0 or 0 => 0
0 or 1 => 1
1 or 0 => 1
1 or 1 => 1
```

D.h. sie liefert immer dann 1, wenn mindestens einer der beiden Werte 1 ist (wobei 1 für jeden im logischen Sinne *wahren* Wert steht, das sind in PERL alle Werte ungleich `Null / undef`). Ist bereits der erste Wert 1, so steht das Ergebnis schon fest, unabhängig vom zweiten Wert: es kommt eine 1 heraus. Dementsprechend bricht or auch nach dem Auswerten des ersten Ausdrucks ab, wenn dieser *wahr* ergibt. Dies kann man sich zunutze machen, indem man zwei PERL-Anweisungen mit `or` verknüpft, auch wenn man gar nicht am Endergebnis der logischen Auswertung interessiert ist. PERL soll in diesen Fällen einfach die erste Anweisung auswerten, und die zweite nur für den Fall, dass die erste Anweisung `Null` (oder `undef`) zurückgegeben hat. Das Auswerten einer Anweisung geschieht in PERL aber genau dadurch, dass die Anweisung effektiv ausgeführt wird, um den Rückgabewert zu erhalten.

Die bekannteste Verwendung des `or`-Befehls dürfte das Öffnen von Dateien sein:

```
1 open FILE, "daten.txt" or die "Die Datei konnte nicht geoeffnet
    werden!";
```

Die `open`-Anweisung wird in jedem Fall ausgeführt, die `die`-Anweisung nur in dem Fall, dass die erste Anweisung einen Nullwert oder ein `undef` zurückgeliefert hat – was sie genau dann tut, wenn etwas schiefgelaufen ist. Läuft beim Öffnen der Datei alles glatt, gibt die `open`-Anweisung nämlich einen positiven Wert zurück und die `die`-Anweisung wird nicht mehr ausgeführt.

Zurück zum Sortieren

Für das Sortieren bedeutet das, dass wir einfach mehrere Abgleiche von `$a` und `$b` mit `or` verknüpfen können. Liefert der erste Abgleich `Null` zurück (was inhaltlich bedeutet, dass `$a` und `$b` in Bezug auf das erste Sortierkriterium gleich sind), dann (und nur dann) wertet `or` den zweiten Abgleich aus.

Um also die geraden Zahlen unter sich und die ungeraden Zahlen unter sich aufsteigend zu sortieren, würde man folgendes schreiben (der Übersichtlichkeit halber in einer Subroutine):

```
1 my @liste = (3, 2, 7, 8, 10, 5);
2 print join(", ", sort multiple_sort @liste), "\n";
3
4 sub multiple_sort {
5
6  return ( (($a % 2) <=> ($b % 2))   or   ($a <=> $b));
7
8 }
```

Diese Kette kann man beliebig fortsetzen für den Fall, dass auch der zweite Abgleich `Null` ergibt.

5.3 Übungsaufgaben

Aufgabe 5.1 Schreibe ein Programm, das die Liste (1, 17, 3, 2) einmal numerisch aufsteigend und einmal numerisch absteigend ausgibt.

Aufgabe 5.2 Schreibe ein Programm, das die Liste `qw(diese woerter will ich sortieren)` einmal alphabetisch aufsteigend und einmal alphabetisch absteigend ausgibt.

Aufgabe 5.3 Schreibe ein Programm, das die Liste `qw(diese woerter will ich sortieren)` sortiert nach der Länge der Wörter ausgibt, und zwar einmal vom kürzesten bis zum längsten Wort, und einmal vom längsten bis zum kürzesten Wort. (Die Abfrage der Länge eines Strings erfolgt mit `length($string)`).

Aufgabe 5.4 Wandle das Beispiel-Programm in Abschnitt 5.2 so ab, dass zuerst alle ungeraden, und dann alle geraden Zahlen ausgegeben werden, wobei die Zahlen innerhalb ihrer Gruppe jeweils absteigend nach ihrem Wert sortiert werden.

Aufgabe 5.5 Schreibe ein Programm, das eine Liste mit Wörtern absteigend sortiert nach der Länge der Wörter ausgibt, wobei gleich lange Wörter unter sich aufsteigend alphabetisch sortiert werden sollen.

6

Anonyme Referenzen

Referenzen müssen zwar immer auf Variablen zeigen, diese Variablen müssen aber nicht immer einen Namen haben. Stattdessen können sie auch anonym im Speicher stehen und nur über die Referenz zugänglich sein. Der Begriff „Anonyme Referenzen“ sollte also eigentlich genauer heißen „Referenzen auf anonyme Variablen“. Diese Referenzen legt man oft an, wenn man ohnehin nur über eine Referenz auf bestimmte Werte zugreifen will. Wozu soll man erst die Werte in einer Variablen speichern und dann eine Referenz auf diese Variable anlegen?

6.1 Wie erzeugt man anonyme Referenzen?

```
1 $array_ref = [ 3, 4, 5 ];
2 $hash_ref = { how => "now", brown => "cow"};
3 $code_ref = sub { Code };     # nur zur Info
4 $scalar_ref = \do{my $anon}; # nur zur Info
```

Statt zu schreiben:

```
1 @array = (3,4,5);
2 $array_ref = \@array; # benannte Referenz
```

kann man also genauso gut schreiben:

```
1 $array_ref = [3,4,5];
```

In beiden Fällen zeigt `$array_ref` auf ein Array mit den Werten 3, 4 und 5. Der Unterschied ist nur, dass dieses Array im ersten Fall nicht nur über die Referenz, sondern auch über einen eigenen Namen, nämlich als `@array`, ansprechbar ist.

6.2 Dereferenzieren von anonymen Referenzen

Für die Dereferenzierung ist es völlig gleichgültig, ob es sich um eine benannte oder um eine anonyme Referenz handelt, in beiden Fällen würde man z.B. schreiben:

```
1 print join(", ", @$array_ref), "\n";
```

oder

```
1 print ${$hash_ref}{how}, "\n"; # oder auch: print $hash_ref->{how},
      "\n";
```

oder

```
1 push(@$array_ref, 9);
```

6.3 Anonyme Kopien

Anonyme Referenzen kann man auch sehr einfach auf Kopien der Inhalte von benannten Variablen anlegen:

```
1 $aref=[ @array ];
2 $href={ %hash };
```

Ergebnis ist (bei der ersten Anweisung) ein Zeiger auf ein Array, das eine Kopie von `@array` darstellt – aber die Kopie ist im Gegensatz zum Original anonym. Die Werte sind im Speicher dupliziert worden, d.h. die Veränderung der Werte im originalen Array beeinflusst nicht die Werte, auf die die Referenz zeigt und umgekehrt. (ganz anders also, als wenn man eine benannte Referenz auf das Array angelegt hätte.)

6.4 Autovivification

Beginnen wir mit einem Beispiel: Man hat in einem Programm nie vorher `$var` angelegt, und schreibt dann plötzlich:

```
1 @$var = (1,2,3);
```

Es handelt sich in diesem Fall nicht um einen abfragenden (lesenden) Zugriff, sondern um eine Zuweisung. Aber eigentlich wird ja nicht direkt `$var` etwas zugewiesen, sondern die einzig sinnvolle Interpretation ist, dass `$var` eine Referenz auf ein Array ist, und genau dem Array, auf das `$var` zeigt, sollen die Werte 1, 2, und 3 zugewiesen werden. Nun ist aber `$var` vorher nie irgendwie angelegt worden, es zeigt auf keinerlei Array, weder benannt noch anonym, demnach gibt es auch keinerlei Array, dem die Werte zugewiesen werden könnten. Man sollte also annehmen, dass diese Anweisung eine Fehlermeldung verursacht.

PERL führt die Anweisung jedoch anstandslos aus. Da die einzige sinnvolle Interpretation der Anweisung ist, dass `$var` eine Referenz auf ein Array enthalten sollte, macht PERL genau dies: es lässt `$var` zu einer Referenz auf ein Array werden. D.h. vom Ergebnis her sind die beiden folgenden Anweisungen äquivalent:

```
1 @$var = (1,2,3);
2
3 $var = [1, 2, 3];
```

Obwohl nur im zweiten Fall wirklich explizit eine Array-Referenz angelegt wird; im ersten Fall wird das Anlegen der Array-Referenz sozusagen implizit verlangt. Genau dieses „Ins-Leben-Rufen“ einer Referenz allein aufgrund des Kontextes und ohne explizite Zuweisung nennt man Autovivification.

Beispiel:

```
1 undef $var;
2 # nur um deutlich zu machen, dass die Variable wirklich vorher noch
      nicht existiert
3 @$var = (1, 2, 3);
4 print $var;
```

Ausgabe: ARRAY(0x80c04f0)

(Das ist PERLs Art zu sagen, dass es sich um eine Array-Referenz handelt, noch gefolgt von der Speicheradresse, wo das Array im Speicher genau liegt).

6.5 Übungsaufgaben

Aufgabe 6.1 Erzeuge ein anonymes Array mit 5 Elementen. Füge 1 Element hinzu. Gib dann das 3. Element aus. Danach gib das ganze Array (am besten mit `join`) aus.

Aufgabe 6.2 Erzeuge 2 anonyme Hashes (mit {...}) und gib dann mit einer `foreach`-Schleife über die beiden Referenzen die Inhalte beider Hashes aus. Das Programm sollte folgende Zeile enthalten:

```
1 foreach my $href ($href1, $href2) {
2 ...
```

Aufgabe 6.3 Erzeuge per Autovivification eine Referenz auf ein anonymes Hash und gib den Inhalt des Hashes aus.

7

Komplexe Datenstrukturen

Ein Array kann nicht nur skalare Werte enthalten; es kann auch weitere Arrays oder auch Hashes als Elemente enthalten. Man spricht dann von einem *Array of Arrays* bzw. von einem *Array of Hashes*, wenn alle Elemente eines Arrays wiederum Arrays sind (bzw. Hashes). Entsprechend können auch die Werte eines Hashes nicht nur Skalare sein, sondern ebenfalls Arrays oder Hashes (*Hash of Arrays / Hash of Hashes*). Man kann diese Strukturen beliebig tief verschachteln (solange man genug Speicherplatz dafür hat ...). Und man kann sie auch beliebig mischen: man kann z.B. ein Array anlegen, bei dem das erste Element ein skalarer Wert ist, das zweite Element ein weiteres Array und das dritte Element ein Hash.

Möglich ist dies, weil PERL dabei intern mit Referenzen arbeitet. Genauer gesagt müsste man nämlich sagen, dass die drei Array-Elemente im Beispiel des gemischten Arrays ein skalarer Wert, eine Array-Referenz und eine Hash-Referenz sind. D.h. alle Werte des Arrays sind intern als skalare Variablen abgelegt.

7.1 Erzeugen von komplexen Datenstrukturen

Hier ein Beispiel, um ein solches gemischtes Array anzulegen:

```
1 $array[0] = 13;
2 $array[1] = [ 3, 7, 9];
3 # Erzeugen und Zuweisen eines anonymen Arrays
4 $array[2] = {colour => "blue", size => "2 feet"};
5 # Erzeugen und Zuweisen eines anonymen Hashes
```

Damit erhalten wir genau die oben angesprochene gemischte Datenstruktur.

Bei einem Hash funktioniert das entsprechend:

```
1 $hash{yellow} = 13;
2 $hash{blue} = [3, 7, 9];
3 # Erzeugen und Zuweisen eines anonymen Arrays
4 $hash{green} = {weight => "20 pounds", size => "2 feet"};
5 # Erzeugen und Zuweisen eines anonymen Hashes
```

Es sollte damit klar sein, wie man ein Array anlegt, dessen sämtliche Elemente weitere Arrays sind (wiederum: genauer gesagt eigentlich Array-Referenzen), und ebenso die anderen erwähnten Datenstrukturen (*Array of Hashes, Hash of Arrays, Hash of Hashes*).

Hier wird auch deutlich, warum man oft anonyme Referenzen verwendet: den untergeordneten Arrays und Hashes will man in diesen Fällen sicher nicht allen einzeln noch Namen geben.

7.2 Ansprechen von komplexen Datenstrukturen

Wenn man nun ein Array of Arrays angelegt hat, wie spricht man dann die inneren Elemente an? Ganz einfach, indem man die Indizes für die verschiedenen Ebenen hintereinander angibt:

```
1 $array[0] = [3, 7, 9];
2 $array[1] = [20, 2];
3 $array[2] = [100, 13, 77, 25, 49, 63, 4, 5];
4
5 print $array[2][7];
```

Dies würde bedeuten: drucke aus demjenigen Array, welches das 3. Element des Basis-Arrays bildet, das 8. Element. (Nicht vergessen, dass die Indizes mit 0 beginnen.)

Da intern, wie gesagt, Referenzen verwendet werden, ist der Ausdruck in der `print`-Anweisung eigentlich nur eine Kurzform für

```
1 $array[2]->[7]
```

Aber die Pfeile kann man sich in diesem Fall sparen.

Dasselbe gilt für Hashes:

```
1 $mitarbeiter{Krause} = { Vorname => "Bernd",
2                          Alter => 33,
3                          Gehalt => 2300
4                        };
5 $mitarbeiter{Mueller} = { Vorname => "Hermann",
6                           Alter => 44,
7                           Gehalt => 3700
8                         };
9 print $mitarbeiter{Krause}{Vorname};
```

Auch das ist eigentlich eine Kurzform von

```
1 print $mitarbeiter{Krause}->{Vorname};
2 # drucke von dem Hash, auf das $mitarbeiter{Krause} zeigt, den Wert
      zum Key "Vorname"
```

Nochmal zurück zu unserem gemischten Array. Hier könnte eine Ausgabe wie folgt aussehen:

```
1 $array[0] = 13;
2 $array[1] = [ 3, 7, 9];
3 $array[2] = {colour => "blue", size => "2 feet"};
4
5 print $array[1][0]; # gibt die 3 aus
6 print $array[2]{colour}; # gibt "blue" aus
7 print $array[2][1]; # verursacht einen netten kleinen Fehler
```

Die letzte Zeile sollte deutlich machen, warum man nicht sehr oft solche gemischten Strukturen verwendet – man verheddert sich zu leicht selbst darin, das wievielte Array-Element von welchem Variablentyp war. Hingegen ist es durchaus manchmal sinnvoll, ein *Hash of Hashes of Hashes* anzulegen:

```
1  $matrikel_nr = 112341234;
2  $noten = {CL1 => "3",
3            CL2 => "4",
4            PERL_fuer_Fortgeschrittene => "1"
5           };
6  $studenten{$matrikel_nr}{Vorname} = "Herbert";
7  $studenten{$matrikel_nr}{Nachname} = "PERLfreak";
8  $studenten{$matrikel_nr}{Noten} = $noten;
9
10 print $studenten{112341234}{Noten}{CL2}, "\n";
```

Bei solchen Strukturen, in denen Referenzen auf Hashes von einem Ausdruck in geschweiften Klammern gefolgt werden, ist es nicht nötig, so etwas wie

```
1 ${$mitarbeiter{$nachname}}{Vorname}
```

zu schreiben oder

```
1 $mitarbeiter{$nachname}->{Vorname}.
```

Die einfache Aneinanderreihung genügt und ist eindeutig:

```
1 $mitarbeiter{$nachname}{Vorname} .
```

(vergleiche demgegenüber `$$x{farbe} = 7;`)

Entsprechendes gilt für Arrays: `$array[3][9]` ist korrekt und eindeutig, es ist nicht nötig so etwas wie `${$array[3]}[9]` oder `$array[3]->[9]` zu schreiben.

7.3 Autovivification und komplexe Datenstrukturen

Da komplexe Datenstrukturen intern auf Referenzen beruhen, kommt uns das PERL-Prinzip der Autovivification sehr zugute: ich kann ohne jegliches vorheriges Anlegen der Struktur einfach schreiben

```
1 $a[3][21][75][2]="fred";
```

`$a[3]` enthält danach automatisch eine Referenz auf ein anonymes Array, dessen 22. Element wiederum eine Referenz auf ein anonymes Array enthält, dessen 76. Element wiederum eine Referenz auf ein anonymes Array enthält, dessen 3. Element mit dem Wert `fred` belegt wird.

Beispiele:

```
1  # hash of arrays:
2
3  $postleitzahlen{Muenchen} = [ 80809, 80358, 80974 ];
4  $postleitzahlen{Koeln} = [ 20270, 20348 ];
5
6  foreach $stadt (keys%postleitzahlen) {
7   foreach $plz (@{$postleitzahlen{$stadt}}) {
8    print "$stadt: $plz\n";
9   }
10 }
11
12 # hash of hashes:
13
14 $mitarbeiter{Krause} = { Vorname => "Bernd",
15                          Alter => 33,
16                          Gehalt => 2300
17                        };
18 $mitarbeiter{Mueller} = { Vorname => "Hermann",
19                           Alter => 44,
20                           Gehalt => 3700
21                         };
22
23 foreach $nachname (keys%mitarbeiter) {
24  print "Name: $nachname, $mitarbeiter{$nachname}{Vorname}\n";
25  print "Alter: $mitarbeiter{$nachname}{Alter}\n";
26  print "Gehalt: $mitarbeiter{$nachname}{Gehalt}\n";
27 }
```

7.4 Übungsaufgaben

Aufgabe 7.1 Belege in einem Array of Arrays den 3. Wert des 2. Arrays mit 7, und den 2. Wert des 1. Array mit 9.

Aufgabe 7.2 Lege einen Hash an, der 5 Schlüssel enthält, wobei jeder Schlüssel eine Zahl ist. Die Zahlen sollen jeweils die Telefonnummern einer WG darstellen. Der Wert zu jedem Schlüssel ist jeweils die Liste der Namen der WG-Bewohner (Implementierung als *Hash of Arrays*, die Telefonnummern sollen die Schlüssel des Hashes sein, die anonymen Arrays mit den Namen der Mitbewohner sollen die zugehörigen Werte sein). Gib den Inhalt des Hashes aus.

Aufgabe 7.3 Lege in einer komplexen Datenstruktur zu verschiedenen Autoren die Titel der von ihnen verfassten Bücher ab (mit mindestens 2 Büchern pro Autor) und zu jedem Buchtitel jeweils noch Erscheinungsjahr, ISBN und Preis. Gib die Daten in übersichtlicher Form aus. (Kleiner Hinweis: mit einem reinen *Hash of Hashes* lässt sich diese Aufgabe nicht mehr lösen, die Struktur muss noch ein klein bisschen komplexer sein.)

8

Sortieren von Hashes und komplexen Datenstrukturen

8.1 Sortieren von Hashes nach Schlüsseln

Sortieren von Hashes nach Schlüsseln funktioniert, indem man sich die Schlüssel mittels `keys %hash` als Liste geben lässt und diese Liste wie gewünscht sortiert. Danach kann man sich dann je nach Bedarf entweder nur die Schlüssel oder auch die Schlüssel mit ihren zugehörigen Werten ausgeben lassen:

```
1 my %hash = (joerg => 7, bernd => 19, susanne => 14, heinrich => 3);
2 foreach my $key (sort { $b cmp $a } keys%hash) {
3     print $key, "\t", $hash{$key}, "\n";
4 }
```

8.2 Sortieren von Hashes nach Werten

Will man nur die Werte selbst haben, braucht man lediglich `values %hash` wie gewünscht zu sortieren und diese Liste dann ausgeben oder weiterbearbeiten. Oft will man aber die Schlüssel in Abhängigkeit von ihren jeweiligen Werten sortieren – z.B. die Namen in dem Hash im ersten Beispiel in Reihenfolge der ihnen zugeordneten Zahlen. Dann sortiert man `keys %hash` und greift in der Sortier-Routine auf die zugehörigen Werte zu:

```
1 my %hash = (joerg => 7, bernd => 19, susanne => 14, heinrich => 3);
2 foreach my $key (sort { $hash{$a} <=> $hash{$b} } keys %hash) {
3     print $key, "\t", $hash{$key}, "\n";
4 }
```

Ist in diesem Beispiel der Schlüssel zu `$a` kleiner als der Schlüssel zu `$b`, so wird `$a` vor `$b` einsortiert. Absteigend sortieren funktioniert dementsprechend mit `sort { $hash{$b} <=> $hash{$a}} .`

8.3 Sortieren von komplexen Datenstrukturen

Dieselben Prinzipien gelten auch für komplexe Datenstrukturen. Sollen z.B. bei folgendem *Hash of Arrays*

```
1 my %postleitzahlen = ();
2 $postleitzahlen{muenchen} = [ 80809, 80358, 80974 ];
3 $postleitzahlen{koeln} = [ 20270, 20348 ];
4 $postleitzahlen{berlin} = [ 34700, 34688, 34701];
```

die Städte in Reihenfolge der ersten ihnen zugeordneten Postleitzahl sortiert werden, dann sähe das Programm z.B. so aus:

```
1 foreach my $stadt (sort { $postleitzahlen{$a}[0] <=>
      $postleitzahlen{$b}[0] } keys %postleitzahlen) {
2   print $postleitzahlen{$stadt}[0], "\t", $stadt, "\n";
3 }
```

Ein weiteres Beispiel: in einem *Hash of Hashes* haben wir festgehalten, wer bei der letzten Party wieviel wovon getrunken hat:

```
1 my %konsumiert = ();
2 $konsumiert{maria}{bier} = 5;
3 $konsumiert{maria}{feiglinge} = 4;
4 $konsumiert{ralf}{cocktails} = 3;
5 $konsumiert{ralf}{bier} = 9;
6 $konsumiert{ralf}{wodka} = 7; # dem ging's am naechsten Morgen
      nicht sehr gut ...
7 $konsumiert{susanne}{osaft} = 2;
8 $konsumiert{susanne}{baileys} = 5;
```

Jetzt wollen wir die Namen alphabetisch ausgeben und darunter jeweils, absteigend nach der Menge sortiert, welche Getränke der / die Betreffende so alles getrunken hat:

```
1 foreach my $gast (sort keys %konsumiert) {
2   print "$gast\n";
3   foreach my $getraenk (sort { $konsumiert{$gast}{$b} <=>
        $konsumiert{$gast}{$a} } keys %{$konsumiert{$gast}}) {
4     print $konsumiert{$gast}{$getraenk}, "\t", $getraenk, "\n";
5   }
6   print "\n";
7 }
```

Ausgabe:

```
maria
5       bier
4       feiglinge

ralf
9       bier
7       wodka
3       cocktails

susanne
5       baileys
2       osaft
```

Wenn man die Ausgabe danach sortieren will, wer am meisten verschiedene Getränke konsumiert hat, vergleicht man, wessen innerer Hash mehr Schlüsselwerte hat: `keys %{$konsumiert{$gast}}` enthält die Liste der verschiedenen Getränke. Wie jede andere Liste auch gibt dieser Ausdruck in skalarem Kontext (wie ihn u.a. <=> erzeugt) die Anzahl der Elemente in der Liste zurück.

Man würde also zunächst die Gäste nach Anzahl der verschiedenen konsumierten Getränke sortieren:

```
1 foreach my $gast (sort { keys %{$konsumiert{$b}} <=> keys %{
      $konsumiert{$a}}  } keys %konsumiert) {
```

und danach z.B. in alphabetischer Reihenfolge die Liste der Getränke ausgeben, jeweils gefolgt von der Menge:

```
1   print "\n$gast:\n\n";
2   foreach my $getraenk (sort keys %{$konsumiert{$gast}}) {
3     print $getraenk, "\t", $konsumiert{$gast}{$getraenk}, "\n";
4   }
5 }
```

Ausgabe:

```
ralf:
bier    9
cocktails       3
wodka   7

susanne:
baileys 5
osaft   2

maria:
bier    5
feiglinge       4
```

Sollen Gäste, die gleich viel verschiedene Getränke konsumiert haben, in alphabetischer Reihenfolge ausgegeben werden (d.h. so, dass Maria vor Susanne ausgegeben wird), würde man die erste `foreach`-Anweisung wie folgt ändern:

```
1  foreach my $gast (sort { (keys %{$konsumiert{$b}} <=> keys %{
       $konsumiert{$a}}) or ($a cmp $b)  } keys %konsumiert) {
```

8.4 Nicht verwechseln!

Es sollte darauf geachtet werden, immer zu unterscheiden **was** sortiert werden soll und **wonach** sortiert werden soll. **Was** sortiert werden soll, sollte immer nach dem sort und der Sortier-Subroutine stehen. Sollen z.B. bei dem Party-Beispiel die Personen durchgegangen werden, dann muss am Ende immer `keys %konsumiert` stehen, und nichts von Getränken oder ähnlichem, auch wenn ich nach Getränken sortieren will. Die Schleifenvariable sollte dementsprechend `$person` o.ä. heißen.

In der Sortier-Subroutine hingegen geht es vor allem darum, **wonach** sortiert werden soll, da müssen die entsprechenden Elemente angesprochen und verglichen werden (z.B. `keys %$konsumiert{$b} <=> keys %{$konsumiert{$a}`, wenn ich die Anzahl der verschiedenen von einer Person konsumierten Getränke vergleichen will). Das **was** sortiert wird muss in den entsprechenden Ausdrücken immer als `$a` bzw. `$b` auftauchen (in diesem Beispiel die Personen; verglichen wird dann z.B. `keys %{$konsumiert{ralf}}` mit `keys %{$konsumiert{maria}}`, d.h. wie viele verschiedene Getränke Ralf getrunken hat vs. wie viele verschiedene Getränke Maria getrunken hat).

8.5 Übungsaufgaben

Aufgabe 8.1 Füge zu dem Mitarbeiter-Hash aus Kapitel 7 noch 2 Einträge hinzu. Gib dann die Mitarbeiter aus, einmal alphabetisch nach Nachnamen sortiert, einmal absteigend nach Gehalt sortiert.

Aufgabe 8.2 Gib das *Hash of Arrays* aus Aufgabe 6.2 zu komplexen Datenstrukturen aus, sortiert nach Anzahl der WG-Bewohner. Bei gleich vielen Bewohnern soll nach Telefonnummer sortiert werden.

Aufgabe 8.3 Gib die Daten aus Aufgabe 6.3 zu komplexen Datenstrukturen wie folgt aus: zuerst jeweils der Autor, absteigend nach Zahl der veröffentlichten Bücher. Wenn zwei Autoren gleich viele Bücher veröffentlicht haben, dann alphabetisch nach dem Namen des Autors. Zu jedem Autor sollen dann alle seine Bücher aufgelistet werden, chronologisch nach Erscheinungsjahr sortiert.

9

Einlesen

9.1 Einlesen von Daten

@ARGV

Dieses Array enthält alle Argumente, die beim Aufruf des Programms in der Kommandozeile mitgegeben worden sind.

Ist das Programm z.B. aufgerufen worden mit

```
perl mein_programm.perl datei1.txt datei2.txt
```

dann enthält `@ARGV` die Liste `qw(datei1.txt datei2.txt)`. Der Name des PERL-Programms selbst taucht hingegen nicht in `@ARGV` auf.

Der Angle Operator (<>)

Der Angle Operator liest stückchenweise Daten aus einem Datei-Handle, z.B.

```
1  while (<IN>) {
2  ....
3
4  }
```

Gelesen wird jeweils bis zum nächsten Vorkommen des Strings, der in `$/` abgelegt ist (*per default* `\n`). Am Dateiende liefert der Angle Operator den Wert `undef` zurück (und `undef` ist logisch *falsch*); somit kann man sehr praktisch eine `while`-Schleife mit dem Angle Operator benutzen.

Auch leere Zeilen sind logisch *wahr*: sie enthalten ja noch den Zeilenumbruch. Beim Lesen von `STDIN` gerät man daher leicht in eine Endlosschleife; ein Abbruch seitens des Benutzers ist möglich mit `CTRL-D` (= Ende der Eingabe, danach wird der Rest des Programms noch ausgeführt) oder `CTRL-C` (beendet das gesamte Programm sofort). Aber natürlich ist es nicht sehr schön, auf dieses Wissen seitens des Benutzers zu bauen. Vom Programm her kann man diesen Fall z.B. wie folgt abfangen:

```
1  while(<>) {
2    chomp;
3    last unless $_;
4    ....
5
6  }
```

$_ und der Angle Operator

Wenn der Angle Operator der einzige Operator in einer `while`-Bedingung ist, und nur dann, wird der jeweils eingelesene String automatisch an `$_` übergeben.

@ARGV und der Angle Operator

Wenn im Angle Operator kein Dateihandle angegeben ist, (also nur <> geschrieben wird), dann öffnet der Angle Operator alle Elemente in `@ARGV` als Dateien, und liest sie aus (bei mehreren Dateien so, als würde es sich um eine einzige Datei handeln). `@ARGV` enthält beim Programmstart alle Argumente, mit denen das Programm in der Kommandozeile aufgerufen wurde. Argumente, die keine Dateien darstellen, müssen also vor Verwendung von <> aus `@ARGV` entfernt werden. Ist `@ARGV` leer, so liest <> von der Datei -, das bedeutet Lesen von `STDIN`. Beim Abarbeiten von `@ARGV` durch <> wird der Inhalt von `@ARGV` gelöscht; `$ARGV` enthält jeweils die aktuell bearbeitete Datei. (siehe auch "Programming PERL", Kapitel 2)

Ein spezielles Datei-Handle: DATA

Manchmal kann es nützlich sein, weder aus einer Datei noch von `STDIN` zu lesen: man will immer mit denselben Daten arbeiten, dafür aber nicht eine Extra-Datei anlegen, sondern diese Daten im Programm selbst speichern. Eine praktische Lösung für diesen Fall ist das Datei-Handle `DATA`. Man beendet das Programm mit der Zeile `__DATA__` und listet danach die einzulesenden Daten auf, so wie man es in einer unabhängigen Datei machen würde.

Beispiel:

```
1  #!/usr/bin/perl -w
2  #Programm data
3  use strict;
4
5  {
6    while (<DATA>) {
7        chomp;
8        print "Gelesen: $_\n";
9    }
10 }
11
12 __DATA__
13 12
14 20
15 37
```

9.2 Einlesen von Kommandozeilen-Optionen

Oft gibt man einem Programm beim Aufruf Optionen mit, um bestimmte Verhaltensweisen des Programms zu bewirken, so z.B. `-h` beim Kommando `ls`, damit die Dateigrößen nicht in Bytes, sondern je nach Größe auch in Mega- oder Gigabyte ausgegeben werden (`h` wie *human readable*), oder `-nr` bei `sort`, damit numerisch absteigend sortiert wird. Im aufgerufenen Programm (ob Unix-Kommando, PERL-Programm oder noch etwas anderes) müssen diese Optionen natürlich irgendwie eingelesen werden, um dann berücksichtigt werden zu können.

@ARGV

Da `@ARGV` automatisch alle Kommandozeilen-Argumente einliest, kann man natürlich daraus die Optionen auslesen. Dies wird aber schnell recht aufwendig, vor allem weil manche Optionen ihrerseits wieder Argumente nehmen, und weil man dauernd zwischen Dateinamen und Optionen unterscheiden muss. Da gibt es bequemere Wege ....

-s

Der wohl einfachste Weg ist das Anfügen von `-s` an die shebang-Zeile (also z.B. `#!/usr/bin/perl -s`). Alle Kommandozeilen-Argumente, die mit einem Bindestrich anfangen, werden dann in Variablen umgewandelt. Ein `-g` beim Aufruf des Perl-Programms z.B. bewirkt dann, dass im Programm die globale Variable `$main::g` ins Leben gerufen und mit dem Wert 1 belegt wird. Bei Benutzung von `use strict` kann man die Variable jedoch nicht einfach als `$g` abrufen, sondern nur als `$main::g`. Dies kann man einfach durch ein `our $g;` ändern – wie bei jedem Aufruf von `our` wird dann im Speicher für lexikalische Variablen unter dem Namen `$g` ein Verweis auf die globale Variable selben Namens im aktuellen Package angelegt, d.h. in diesem Fall auf `$main::g`. Es bietet sich an, die `our`-Zeile auch gleich zur Kommentierung der Funktion zu nutzen:

```
 1 #!/usr/bin/perl -s
 2
 3 use strict;
 4 use warnings;
 5
 6 # Erlaubte Kommandozeilen-Optionen:
 7
 8 our $g; # Ausgabe in Grossbuchstaben
 9 our $k; # Ausgabe in Kleinbuchstaben
10
11 # Hauptprogramm:
12
13 while (<DATA>) {
14   if ($g) {
15     print uc($_);
16   } elsif ($k) {
17     print lc($_);
18   } else {
19     print $_;
20   }
21 }
22
23
24 __DATA__
25 Hallo
26 User
```

Die Optionen können problemlos auch aus mehreren Buchstaben bestehen, z.B. würde das Argument `-gross` im Programm zum Anlegen der globalen Variablen `$main::gross` führen (und ihrer Belegung mit dem Wert 1). Und man kann den Optionen auch Werte mitgeben, angefügt durch ein Gleichheitszeichen, so führt z.B. das Kommandozeilen-Argument `-min=5` zum automatischen Anlegen der Variable `$main::min`, die mit dem Wert 5 belegt ist.

Was hingegen nicht möglich ist, ist das Zusammenfassen mehrere Optionen, wie z.B. `-nr` beim Unix-Kommando `sort`. Dies würde nicht als Aktivierung der beiden Optionen `-n` und `-r` verstanden, sondern es würde die globale Variable `$main::nr` gesetzt und mit dem Wert 1 belegt. `-s` löscht im übrigen die als Option interpretierten Kommandozeilen-Argumente aus @ARGV.

Ein Nachteil von `-s` ist, dass der Benutzer entscheidet, welche Variablen wie gesetzt werden; das Programm hat keine Möglichkeiten, das Belegen von globalen Variablen durch den Benutzer zu verhindern. Ruft der Benutzer die überhaupt nicht vorgesehene Option `-x=7` auf, so wird eine globale Variable `$main::x` erzeugt und mit 7 belegt, ob das nun im Programm vorgesehen ist oder nicht, und auch unabhängig davon, ob diese Variable nicht vielleicht eine ganz andere Funktion im Programm hat.

Getopt::Std

Eine Alternative zu `-s` ist das Modul `Getopt::Std`. Es erlaubt nur Ein-Buchstaben-Optionen, mit oder ohne Wert. Der Wert muss mit einem Leerzeichen getrennt hinter der Option folgen, nicht mit einem Gleichheitszeichen wie bei `-s` (also z.B. `-m 5` statt `-min=5`).

Damit das Modul zur Verfügung steht, muss man es zunächst einbinden:

```
1 use Getopt::Std;
```

Danach kann man dann irgendwann entweder die Funktion `getopt()` oder die Funktion `getopts()` aufrufen.

getopt

`getopt()` erwartet grundsätzlich Argumente – es ist nicht möglich, einfach nur eine Option ohne Argument aufzurufen. Hier ein Beispiel für das Parsen der Optionen mit `getopt()`:

```
1 getopt("kg");
```

Dies bedeutet: wenn die Kommandozeilen-Optionen `-k <Argument>` und/oder `-g <Argument>` aufgerufen wurden, werden im Programm die Variablen `$opt_k` bzw. `$opt_g` auf die entsprechenden Werte gesetzt. Alternativ kann man auch eine Hash-Referenz als zweites Argument mitgeben:

```
1 getopt("kg", \%optionen);
```

Dann werden nicht die Variablen `$opt_k` bzw. `$opt_g` erzeugt und belegt, sondern die Hash-Werte `$optionen{k}` und `$optionen{g}`.

getopts

Die Variante `getopts()` funktioniert wie `getopt()`, erlaubt aber sowohl Boole'sche Optionen (d.h. ohne Argumente, im Programm dann entweder 0 oder 1 gesetzt) als auch Optionen mit Argumenten. Optionen, bei denen Argumente erwartet werden, müssen jeweils durch einen Doppelpunkt hinter dem Buchstaben der Option gekennzeichnet werden. So erlaubt z.B.

```
1 getopts("k:g");
```

ein Argument bei der Kommandozeilen-Option `-k`, aber kein Argument bei der Option `-g`.

Auch hier kann man eine Hash-Referenz mitgeben, in der dann die entsprechenden Werte belegt werden.

Bei beiden Methoden werden Optionen bzw. Argumente automatisch zurückgewiesen, wenn sie nicht im String angegeben sind (d.h. es werden Warnmeldungen ausgegeben).

Zusammenfassung

Vorteile von Getopt::Std gegenüber "-s":

- Man hat im Programm die volle Kontrolle darüber, welche Optionen und Argumente zulässig sind; der Benutzer kann keine weiteren globalen Variablen setzen, die programm-intern evtl. zu Konflikten führen könnten.
- Der Benutzer wird automatisch darauf aufmerksam gemacht, wenn er eine nicht vorgesehene Option aufruft.
- Die Namen der automatisch angelegten Variablen machen einen Konflikt mit anderen Variablen unwahrscheinlicher; man kommt leichter in Versuchung, irgendwo im Programm nochmal `$k` in anderer Bedeutung zu verwenden als `$opt_k` oder gar `$optionen{k}`

- Das Clustern von Optionen ist für den Benutzer möglich.

Nachteile:

- Die Optionen dürfen jeweils nur einen Buchstaben lang sein.
- Bei `getopt()` müssen immer Argumente mitgegeben werden, reine Boole'sche Optionen funktionieren nicht.

Getopt::Long

Eine weitere Alternative ist die Verwendung des Moduls `Getopt::Long`. Hier können die Optionen mehrere Buchstaben lang sein; sie müssen jeweils mit einem doppelten Bindestrich eingeleitet werden, z.B. `--verbose`.

Die Funktion, die die Optionen parst, heißt `GetOptions`; sie muss mit einem Hash als Argument aufgerufen werden, bei dem die Schlüssel den erlaubten Optionen entsprechen und die Werte Referenzen auf skalare Variablen sein müssen. Alles in allem sieht das dann z.B. so aus:

```
1 use Getopt::Long; # Einbinden des Moduls
2
3 GetOptions( "gross" => \$grossbuchstaben,
4             "klein" => \$kleinbuchstaben, "egal" => \$egal);
```

Damit werden in diesem Beispiel dann die Variablen `$grossbuchstaben`, `$kleinbuchstaben` und `$egal` auf 1 bzw. 0 gesetzt, je nachdem, ob beim Aufruf die Optionen `--gross`, `--klein`, `--egal` angegeben worden sind oder nicht.

An die Namen der Optionen kann man noch verschiedene Zusätze anhängen. Hängt man keine Zusätze an, wie im Beispiel, dann dürfen die Kommandozeilen-Argumente keine Argumente haben, die entsprechenden Variablen werden entweder auf 0 oder auf 1 gesetzt.

Folgende Zusätze sind möglich:

`!` Die Option kann auch mit vorangestelltem `no` auf der Kommandozeile eingegeben werden, z.B. würde `GetOptions("sort!"=> \$sort);` die Kommandozeilen-Optionen `--sort` und auch `--nosort` akzeptieren.

`=s` Ein obligatorisches String-Argument muss folgen, z.B. `GetOptions("filename=s"=>\$file);`

`:s` Ein String-Argument ist erlaubt, aber nicht obligatorisch.

`=i` Integer-Argumente.

`:i` Integer-Argumente.

`=f` Floating-point-Argumente (z.B. 3.141).

`:f` Floating-point-Argumente (z.B. 3.141).

Bei `Getopt::Long` können die Argumente von Optionen entweder durch ein Gleichheitszeichen oder durch ein Leerzeichen von der Option getrennt sein.

Vorteile von Getopt::Long gegenüber Getopt::Std:

- Die Optionen können mehrere Buchstaben lang sein und sind damit aussagekräftiger
- Die Argumente von Optionen können genau gesteuert werden: sie können verboten, erlaubt oder obligatorisch sein und sie können auf Strings, Integers und Floating-Point-Zahlen eingeschränkt werden.

Nachteile:

- Alle Argumente müssen mit zwei Bindestrichen eingeleitet werden
- Auch sonst etwas mehr Schreibarbeit
- Die Möglichkeit, mittels einer Hash-Referenz ein Hash für die Optionen anzulegen entfällt.
- Das Clustern von Optionen ist nicht möglich.

Hilfe!

Jedes Programm, dass mit Optionen arbeitet, sollte entweder die Option `-h` oder `--help` verstehen und bei Aufruf dieser Option eine Übersicht über die möglichen Optionen ausgeben. Dasselbe sollte passieren, wenn obligatorische Optionen/Argumente fehlen oder sonstwie festgestellt wird, dass in Bezug auf die Optionen etwas nicht Zulässiges/Sinnvolles eingegeben wurde.

9.3 Übungsaufgaben

```
1  #!/usr/bin/perl -w
2
3  use strict;
4
5  my $b = "";
6  while(<>) {
7    chomp;
8    $b = reverse($_);
9    print "$b\n";
10 }
```

Aufgabe 9.1 Erweitere das Programm so, dass zusätzlich vor jeder umgedrehten Eingabezeile der Name der gerade bearbeiteten Datei ausgegeben wird.

Aufgabe 9.2 Ergänze das Programm so, dass es abbricht, falls von `STDIN` gelesen wird und vom Benutzer eine Leerzeile eingeben wird (aber nicht, falls aus einer Datei eingelesen wird und darin eine Leerzeile auftritt)

Aufgabe 9.3 Erweitere das Programm so, dass es nach der `while`-Schleife die Namen aller bearbeiteten Dateien ausgibt (Jeder Dateiname sollte in der Ausgabe nur einmal vorhanden sein).

Aufgabe 9.4 Wandle das ursprüngliche Programm so ab, dass die Daten nicht von `STDIN` oder einer externen Datei gelesen werden, sondern aus der Programmdatei selbst.

Aufgabe 9.5 Schreibe ein Programm, dass unter `__DATA__` mehrere Zeilen mit Zahlen enthält. Im Programm sollen diese Zahlen sortiert werden. Der Benutzer soll über Optionen steuern können, ob die Zahlen alphabetisch oder numerisch sortiert werden sollen, und ob sie auf- oder absteigend sortiert werden sollen. Außerdem soll er ein Minimum angeben können; wenn eine Zahl kleiner ist als dieses Minimum, soll sie nicht mit ausgegeben werden. Realisiere das Programm mit `-s`. Dabei jeweils die Hilfs-Option nicht vergessen (Ausgabe über `STDERR`), und auch nicht die Hinweise zum Schreiben von Programmen!

10

Unicode und Kodierung

10.1 Was ist ein Zeichensatz?

Wenn Daten übertragen, gespeichert oder eingelesen werden, geschieht das meist auf der Basis von Bytes. Ein Byte ist eine Folge von acht Bits, die wiederum die Zustände Eins und Null annehmen können. Ein Byte kann also 265 ($= 2^8$) verschiedene Zustände annehmen. Es ist praktisch, Bytes in Hexadezimalschreibweise anzugeben. So kann man mit zwei Ziffern genau ein Byte darstellen, und eine Ziffer bezeichnet genau vier Bit. Zeichensätze (engl.: *codesets*) sind Standards, die festlegen, welches Zeichen einem Byte (oder mehreren) zuzuordnen ist. Zeichen sind in der Regel Buchstaben, Zahlen, aber auch die *Kontrollzeichen* wie zum Beispiel *newline* und *escape*. Auch enthalten sie keine Informationen darüber, wie die Zeichen auszusehen haben; dafür sind die Schriftsätze (*Fonts*) zuständig. Zeichensätze werden dann als Kodierung (engl.: encoding) bezeichnet, wenn der Computer das Zeichen aus seiner Umgebung berechnen muss.

10.2 Was ist Unicode?

Unicode ist ein Standard, der vom Unicode-Konsortium festgelegt wird. Das Unicode-Konsortium ordnet jedem Zeichen einen bestimmten *Codepoint* und einen *Namen* zu. Diese Codepoints sind Zahlen, die in Hexadezimalschreibweise angegeben werden. Die Namen sind eindeutig bestimmte Bezeichnungen in Großbuchstaben. So kann jedes Zeichen kodierungsunabhängig identifiziert werden. Um im PERL-Code Unicode-Zeichen anzugeben, gibt es folgende Möglichkeiten:

- `"\x{61}" # ein a` mit `\x{Zahl}` kann man den Codepoint angeben, bei weniger als drei Ziffern kann man die Klammern auch weglassen.
- `"\N{UMBRELLA}" # ein Regenschirm` mit `\N{Name}` kann man den Unicode Namen direkt eingeben. Um `\N` verwenden zu können muss man vorher `use charnames ":full";` aufrufen.
- Wenn man `use utf8;` angibt kann man UTF-8–Zeichen direkt in den Programmcode eingeben. Man sollte aber sicherstellen, dass der Programmcode auch in UTF-8 kodiert ist.

Zusätzlich stellt das Unicode-Konsortium diverse Umwandlungsformate von Codepoints zu Bitfolgen (Unicode Transformation Format, kurz UTF) bereit. Unter anderem gibt es noch Sortierungs- und Großschreibungs-Algorithmen (auch für einzelne Sprachen, was von PERL zum größtem Teil über `use locale` unterstützt wird).

Byte-Order-Mark

Der Byte-Order-Mark ist eine Bytefolge, die am Anfang der Datei stehen oder am Anfang der Datenübertragung gesendet werden kann, um mitzuteilen, welches UTF verwendet wird. Es ist immer die Bytefolge, die für den Unicode-Codepoint `FEFF` steht. Wird dieser „falsch“ – also das/die hintere(n) Byte(s) zuerst – gelesen, bekommt man `FFFE` zurück, das als „not a Character“ definiert ist. Der Byte-Order-Mark ist vor allem für UTF-16 und UTF-32 von Bedeutung, um bei ihnen die

Byte-Reihenfolgen *little endian* (niedrigwertigstes Byte zuerst) und *big endian* (höchstwertigstes Byte zuerst) zu unterscheiden.

Einige Kodierungen und Zeichensätze

- **ASCII:** Alter Zeichensatz, der nur 7 Bit verwendet und 128 verschiedene Zeichen definiert; die meisten Kontrollzeichen sind enthalten, allerdings keine Umlaute. UTF-8, und Latin-1 sowie einige andere Kodierungen enthalten den ASCII-Zeichensatz, verwenden aber das achte Bit, um weitere Zeichen zu kodieren.
- **Latin-1** (auch ISO-8859-1): Jedem möglichen Byte ist ein Zeichen zugeordnet; das ergibt 256 verschiedene Zeichen; enthält Zeichen, um westeuropäische Sprachen einschließlich Deutsch darzustellen.
- **UTF-8** ist eine Kodierung, mit der sich der gesamte Unicode-Zeichenvorrat kodieren lässt. UTF-8 verwendet ein Byte für den ASCII-Bereich, ansonsten mehr.
- **UTF-16/UTF-32** sind jeweils zwei Kodierungen, die noch weiter in *Little Endian (LE)* und *Big Endian (BE)* unterteilt werden, je nachdem, ob das niedrigwertigste Byte zuerst kommt (z.B. Intel x86-Architektur) oder das höchstwertige Byte (z.B. einige IBM-Großrechner). Ein Zeichen enthält mindestens zwei (UTF-16) bzw. vier Byte (UTF-32).
- **UTF-7:** Eine weitere Unicode-Kodierung, die alle Unicode-Codepoints in Zeichen abbilden kann; das erste Bit wird aber nicht genutzt. Wird dort verwendet, wo das Erste Bit eine Kontrollfunktion hat.

Diese Liste lässt sich natürlich noch beliebig fortsetzen. Erwähnenswert wäre noch, dass es noch ISO-8859-2 bis ISO-8859-16 gibt, die jeweils den ASCII um eine oder mehrere Sprachen erweitern, sowie diverse andere Kodierungen, um asiatische Sprachen darzustellen.

10.3 Einlesen und Schreiben von Daten in verschiedenen Kodierungen

PERL speichert Buchstaben-Daten intern (in der Regel) als UTF-8. Wenn PERL nicht weiß, wie die Daten kodiert sind, die es einliest (z.B.: mit dem <> -Operator), geht es davon aus, dass es reine Byte-Dateien sind und behandelt sie anders. Es ändern sich fast[1] alle Funktionen, die mit Strings arbeiten, sowie die Klassen in den regulären Ausdrücken.

Die -C Kommandozeilenoption

Dies gilt ab **PERL-Version 5.8.1** (Ausgabe der PERL-Version: `ich@meinComputer$: perl -v`); vorher wurde diese Option nur auf Windows-Systemen für etwas anderes benutzt!

Das C-Flag sagt PERL welche Input- und Output Streams in UTF-8 kodiert sind. Meistens (vor allem auf UTF-8 basierenden Systemen) wird es ohne Parameter aufgerufen; dann bedeutet es, dass von/auf alle IO-Streams mit UTF-8 gelesen/geschrieben wird, solange die `locale` auf UTF-8 gesetzt ist.

Das C-Flag kann nur beim Aufruf benutzt werden, nicht in der shebang-Zeile. Will man seine Auswahl einschränken, kann man das entweder mit einer Zahl tun (addiert man zwei Optionen, so gelten beide) oder mit Buchstaben (`perl -CIO` ist dasselbe wie `perl -C3`). Will man keine dieser Möglichkeiten benutzen, kann man mit `-C0` alle abschalten (auch solche, die evtl. durch die Environment-Variable `PERL_UNICODE` gesetzt wurden).

[1] `vec()`, `pack`, `unpack` und alle Befehle, die direkt mit dem System kommunizieren, arbeiten NICHT mit UTF-8

Möglich sind folgenden Parameter:

I	1	STDIN wird als UTF-8 gelesen
O	2	auf STDOUT
E	4	auf STDERR wird in UTF-8 geschrieben
S	7	IOE
i	8	:utf8 ist ein Standard-PERL-IO-Layer (s.u.) für Input-Streams
o	16	:utf8 ist ein Standard-PERL-IO-Layer (s.u.) für Output-Streams
D	24	io
A	32	@ARGV wird für UTF-8 gehalten (Vorsicht auf Windows Rechnern!)
L	64	Die bis jetzt genannten (Nummer kleiner als 64) werden nur angewendet, wenn eine der Locale-Variablen `LC_ALL`, `LC_TYPE` oder `LANG` auf UTF-8 gesetzt sind, (Konditionale Ausführung)
a	256	Im Debug-Mode wird UTF-8 Code ausgeführt.

IO-Layers

Wenn man Daten in verschiedenen Kodierungen hat, oder andere als UTF-8 benutzen will, kann man das leider nicht mehr (komfortabel) über die Kommandozeile regeln. PERL verwendet zum Ein- und Auslesen so genannte IO-Layer. Diese bestimmen wie PERL die Daten, die es einlesen soll, verändert, (und ob sie als "Buchstaben-Daten" markiert werden sollen). Mittels dieser Layer lässt sich auch mitteilen, in welcher Kodierung die Daten vorliegen.

Hat man eine Datei in einer bestimmten Kodierung und möchte diese auslesen, kann man mit `open` in der Drei-Argument-Schreibweise oder mit `binmode` in der Zwei-Argument-Schreibweise die nötigen Layer angeben. Man schreibt einfach hinter das Richtungszeichen (das man bei `open` nicht weglassen darf!), ob zum Lesen oder Schreiben geöffnet wird. Mit `:Layer` bestimmt man, welches Layer verwendet werden soll.

:encoding(X)

Um PERL mitzuteilen, in welcher Kodierung die Daten vorliegen, kann man diese Layer benutzen. Für X ist natürlich noch die gewünschte Kodierung einzufügen.

Beispiel:

```
1   open IN , "<:encoding(latin1)", "datei1.txt";
2   open OUT, ">:encoding(UTF-8)", $datei;
3   binmode IN, ":encoding(koi8)"
```

:utf8

Damit gibt man an, dass die Daten im PERL-internen Zeichensatz vorliegen (oder geschrieben werden sollen). Dieser ist meistens UTF-8[2]. Alle Zeichen, mit denen PERL umgehen kann, können damit gedruckt werden.

:locale

PERL sieht beim Betriebssystem in den Environment-Variablen `CODESET`, nach Kodierungen danach in `LC_ALL` und `LANG`. Wenn nichts gefunden wurde sucht PERL in den letzten beiden nach „etwas das wie UTF-8 Aussieht" (perldoc), wird es fündig, wird `:utf8` als Standard gesetzt.

[2] Ausgenommen sind Betriebssysteme, die mit dem EBCDIC-Standard arbeiten; dort ist es UTF-EBCDIC

:crlf

Konvertiert alle Windows-Zeilenenden in \n; wird auf Windows-Rechnern normalerweise immer benutzt.

use open

Mit `use open` und entsprechenden Anweisungen lassen sich IO-Layer für das gesamte Programm festlegen.

Verwendung:

- `use open IN => ":encoding(UTF-8)";` Input-Streams werden mit `UTF-8` geöffnet
- `use open OUT => ":encoding(latin1)";` Output-Streams werden mit `latin1` geöffnet
- `use open IO => ":encoding(koi8)";` Input- und Output-Streams werden mit `:encoding(koi8)` geöffnet; ist äquivalent zu `use open ":encoding(koi8)";`.
- `use open ":std";` ist zusätzlich zu anderen `use open`-Statements anzugeben. Für `STDIN`, `STDOUT` und `STDERR` gelten nun auch die entsprechenden Layer. (`STDIN` ist ein `IN`-Stream `STDOUT` und `STDERR` sind `OUT`-Streams).

use utf8::all

Eine Möglichkeit, UTF-8 umfassend als Default zu definieren, bietet das Pragma `use utf8::all`. Es sorgt dafür dass das PERL-Programm, die Programmargumente, und alle Ein- und Ausgabedateien einschließlich STDIN, STDOUT und STDERR als UTF-8-kodiert interpretiert werden.

Hinweise:

- Die Unicode-Seite: `http://www.unicode.org`
- Wenn man ein bestimmtes Zeichen sucht: `http://www.unicode.org/standard/where`
- Falls man die Kodierung einer Datei bestimmen will, ist oft das UNIX-Tool `file` eine Hilfe.
- Um mehr Kodierungen verwenden zu können, gibt es das Modul `Encode::Supported` (zur Verwendung von Modulen siehe nächste Kapitel).
- man-pages zu diesem Thema: `codesets`, `unicode` und `ISO-8859`

11

Module

11.1 Module: Aufbau und Einbindung

Module sind Programmteile in getrennten Dateien. Diese Modul-Dateien haben grundsätzlich die Endung `.pm`. Solche Module sind sehr bequem auch in anderen Programmen zu benutzen. Sie enthalten globale Funktionen und Variablen, die man aus dem aufrufenden Programm heraus ansprechen kann.

Einbindung von Modulen:

(a) mit `require` oder

(b) mit `use`

Die beiden Befehle unterscheiden sich in 2 Punkten:

1. Zeitpunkt der Einbindung des Moduls:
   `require` lädt Module erst zur Laufzeit, d.h. sobald der `require`-Befehl erreicht wird. `use` lädt die Module schon beim Kompilieren.

   Vorteil von `require`: Man kann es in Programmteilen benutzen, die eventuell nie erreicht werden – dann spart man sich auch das Laden des Moduls. Dies ergibt in der Praxis allerdings in den seltensten Fällen einen spürbaren Vorteil.

   Vorteil von `use`: Falls ein Modul nicht geladen werden kann, erfährt man es direkt beim Kompilieren und erhält nicht einen Programmabbruch, nachdem das Programm eventuell schon lange gelaufen ist.

2. Import von Variablen:
   `use` importiert automatisch die Namen von Variablen und Funktionen, so dass man sie ansprechen kann, ohne explizit das Package angeben zu müssen. `require` hingegen importiert keine Namen von Variablen oder Funktionen. Man kann das auch so ausdrücken:

   ```
   1  use Module = require Module + Module->import();
   ```

   Der Pfeil bedeutet hier den Aufruf einer Objekt-Methode; mehr dazu später.

Beide Unterschiede zwischen `use` und `require` führen dazu, dass `require` sehr selten benutzt wird und `use` die Standardmethode ist, um Module zu einzubinden.

Sowohl bei Einbindung mit `use` als auch mit `require` wird der Programmcode im Modul bei der Einbindung ausgeführt. Dies hat normalerweise keine direkt sichtbaren Folgen, da in Modulen üblicherweise nur Variablen und Subroutinen deklariert und definiert werden, sie aber darüberhinaus keine auszuführenden Anweisungen enthalten.

Syntax von Aufrufen mit use / require

```
1  require "FileHandle.pm";
2  require FileHandle;
3  # ".pm" wird vorausgesetzt, funktioniert wie 1. Aufruf
4  use FileHandle;
5  # Syntax funktioniert wie die von "require"
6  use Cards::Poker;
```

:: in einem Modul-Namen wird in das jeweilige Symbol des Betriebssystems übersetzt, das in Verzeichnispfaden die Ebenen trennt; unter Unix/Linux z.B. wird das angegebene Modul gesucht als Cards/Poker.pm. Dieses System korrespondiert also sehr gut mit einer geordneten Verzeichnisstruktur für die Module: innerhalb des Modulverzeichnisses auf dem benutzten Rechner kann man z.B. ein Unterverzeichnis Cards anlegen, in dem dann die Module Poker.pm und Doppelkopf.pm liegen; diese kann man dann in einem Programm aufrufen mit use Cards::Poker; bzw. use Cards::Doppelkopf;.

Schreiben eines eigenen Moduls, Import und Export von Variablen

```
1  package Cards::Poker;
2  use Exporter;
3  @ISA = ('Exporter');
4  @EXPORT = qw(&shuffle @card_deck);
5  @card_deck = ();  # hier muesste eine echte Liste stehen
6  sub shuffle {}
7  # hier muesste eine echte Funktionsdeklaration stehen
8     .... # ggfs. weitere Deklarationen
9
10 letzte Zeile: 1;     # nicht vergessen!
```

Zeile 1: der Package-Name: Man gibt dem Modul einen eigenen Namensraum für die Variablen, und zwar genau dementsprechend, wie das Modul mit use angesprochen wird. Also in diesem Fall nicht nur Poker, sondern Cards::Poker. Dies ist eine feste Konvention, die beachtet werden muss:

- So ist immer klar, zu welchem Package die Variablen aus einem Modul gehören und es gibt keine Überschneidungen.
- Andernfalls funktioniert der Variablenimport mit My_module_name->import() nicht!

Zeilen 2 und 3: Nutzung des Exporter-Moduls: hilft, das Modul-Interface zu organisieren

Einbindung von Cards::Poker

'use Cards::Poker' = 'require Cards::Poker' + 'Cards::Poker->import()'

Verwendung von require

Alle globalen Variablen sind ansprechbar, müssen aber voll qualifiziert angesprochen werden (d.h. mit Package-Angabe). import() sorgt dafür, dass die Variablen ins Package des aufrufenden Programms übernommen werden, d.h. statt als Poker::Deck::shuffle(23) kann ich die shuffle-Funktion einfach mit shuffle(23) aufrufen.

Problem: keine import-Funktion im Modul vorhanden ⇒ PERL sucht im Package-spezifischen @ISA (d.h. hier in @Cards::Poker::ISA) danach, wo es eine import-Methode hernehmen kann.

```
1  use Exporter;
2  @ISA = ('Exporter');
```

sagt ihm, dass es das Modul Exporter laden soll und die import-Methode aus dem Modul Exporter verwenden soll. (per Vererbung)

Was soll mit der `import`-Methode importiert werden?

Defaultmäßig alles, was in `@EXPORT` enthalten ist. Das ist das sog. "Public Interface", d.h. diese Variablen und Methoden sind dann ohne Package-Angabe ansprechbar. Alle anderen globalen Variablen des Moduls sind nur mit Package-Angabe ansprechbar, das ist aber normalerweise gar nicht vorgesehen.

Was passiert bei gleichlautenden Variablen oder Funktionen im aufrufendem Programm und im Modul?

Prinzipiell gilt die zuletzt vorgenommene Deklaration. Wird also im aufrufenden Programm eine Variable oder Funktion neu deklariert, die schon aus einem Modul importiert wird, so überschreiben sie die Werte / Deklarationen aus dem Modul. Aber das Überschreiben funktioniert auch umgekehrt: zahlreiche (nicht alle) PERL-eigenen Funktionen können per Import neu definiert (*überladen*) werden!

Obligatorischer Abschluss eines Moduls

... ist ein wahrer Wert, gewöhnlich 1. Dies ist Rückgabewert des `use`- oder `require`-Aufrufs; er verursacht einen Fehler, falls er nicht im Boole'schen Sinne *wahr* ist.

Weitere Variablen im Modul, die `import` beeinflussen:

(a) Versionskontrolle:
`$VERSION` kann z.B. mit `$VERSION = 1.00` belegt werden; beim Aufruf von `use` kann dann ein Minimum für die Versionsnummer festgelegt werden. Ist die im Modul definierte Versionsnummer kleiner als die im `use`-Aufruf genannte, bricht das Programm ab.
Bsp: `use Cards::Poker 1.1;` würde dann zu einem Programmabbruch führen.

(b) optionaler Export:
Hier können Variablen definiert werden, die nicht defaultmäßig importiert werden, bei denen der Export aber trotzdem möglich sein soll.
Bsp:

```
1  @EXPORT = qw (module_sub1 module_sub2 @module_list);
2  @EXPORT_OK = qw (module_sub3 %module_hash);
```

Mögliche Aufrufe:

```
1  use Some::Module;
2  # importiert alles, was in @EXPORT steht:
3  # module_sub1, module_sub2 und @module_list
4
5  use Some::Module qw(module_sub1 @module_list);
6  # importiert nur module_sub1 und @module_list,
7  # nicht aber module_sub2
8
9  use Some::Module qw(module_sub3);
10 # importiert nur module_sub3,
11 # nicht aber den Inhalt von @EXPORT!
12
13 use Some::Module qw(:DEFAULT module_sub3);
14 # importiert den Inhalt von @EXPORT und zusaetzlich module_sub3
15
16 # Aufruf mit Kontrolle der Versionsnummer u. import-Liste:
17 use Some::Module 1.3 qw(module_sub1);
```

Was tun mit Variablen, auf die garantiert nicht zugegriffen werden soll?

⇒ **Lexikalisierung mit** `my`

Sie sind dann auch qualifiziert und nicht von außerhalb des Moduls ansprechbar; im Modul selbst sind sie entsprechend ihrem Gültigkeitsbereich ansprechbar.

Es ist aber nicht unbedingt nötig, alle Variablen zu lexikalisieren – wenn sie nicht dokumentiert sind, wird kaum jemand auf die Idee kommen, sie zu importieren. Und falls doch jemand den Programmcode des Moduls so genau studiert, dass er die Variable importieren will, dann muss man das auch nicht unbedingt verhindern.

11.2 Module mit der CPAN-Shell installieren

Vorhandene Module installieren

Ist ein Modul bereits installiert?

- Versuchen es mit `use` einzubinden ⇒ Fehlermeldung falls nicht installiert
- siehe Programm `pmdesc.perl` aus dem PERL Cookbook, Kap. 12.19, um einen Überblick über bereits installierte Module zu erhalten.

Wohin installiert man Module, wenn man nicht Sysadmin ist?

`use` oder `require` suchen angegebene Module normalerweise in allen Pfaden, die in `@INC` aufgelistet sind. Kontrolle mit

```
perl -e 'for (@INC) {print "$_\n"}'
```

Beispiel-Output:

```
/usr/lib/perl5/5.10.0/x86_64-linux-thread-multi
/usr/lib/perl5/5.10.0
/usr/lib/perl5/site_perl/5.10.0/x86_64-linux-thread-multi
/usr/lib/perl5/site_perl/5.10.0
/usr/lib/perl5/vendor_perl/5.10.0/x86_64-linux-thread-multi
/usr/lib/perl5/vendor_perl/5.10.0
/usr/lib/perl5/vendor_perl
.
```

– Plattform-abhängige und -übergreifende PERL-Module, die standardmäßig mitgeliefert werden
– Plattform-, sowie versionsabhängige zusätzlich installierte PERL-Module
– aktuelles Verzeichnis

Wenn man nicht Sysadmin ist, kann man diese Verzeichnisse aber nicht ansprechen. Also bittet man entweder einen Sysadmin, das gewünschte Modul zu installieren (durchaus sinnvoll, wenn es nicht ein sehr ungebräuchliches Modul ist), oder aber man installiert das Modul in ein privates Modulverzeichnis, das man selbst bestimmt (das kostet einen dann ein bisschen Plattenplatz).

Ansprechen von Modulen in einem privaten Verzeichnis

1. Aufruf von `perl` mit der Option `-I`, gefolgt von Verzeichnissen, wo gesucht werden soll (mehrere Verzeichnisse mit Doppelpunkt trennen). Diese Verzeichnisse werden in `@INC` aufgenommen. Die Aufnahme in die Shebang-Zeile im Programm empfiehlt sich ebenfalls nicht, da die Länge dieser Zeile auf manchen Systemen begrenzt ist.

2. Nutzen des `lib`-Pragmas. (Pragmas sind Module, die sich bereits beim Kompilieren auswirken.)

   Beispiel für den Aufruf des lib-Pragmas: `use lib "/projects/biopath/lib";`

   Nachteil: Der Pfad steht immer noch explizit im Programmcode und muss bei Änderungen angepasst werden.

3. Setzen der Umgebungsvariablen $PERL5LIB im Shell-Konfigurationsfile Angenommen, man installiert alle Module im Homeverzeichnis in `~/perl5/`, dann sollte man in seinem c-shell- oder tc-shell-Konfigurationsfile folgenden Eintrag ergänzen:

   ```
   setenv PERL5LIB ~/perl5
   ```

   Syntax für sh, bash, ksh, zsh:

   ```
   export PERL5LIB=$HOME/perl5
   ```

   (Vorsicht! Keine Leerzeichen vor oder nach dem Gleichheitszeichen einfügen!)

Das c-shell-Konfigurationsfile befindet sich normalerweise als `.cshrc` im Homeverzeichnis; der Eintrag sollte an einer Stelle ergänzt werden, die nicht von einer `if`-Bedingung abhängig ist. Vorsicht, falls ihr einen Account bei den Informatikern der LMU habt, dann habt ihr im Normalfall eine z-shell, und entsprechend ein Konfigurationsfile namens `.zshrc` in eurem Homeverzeichnis. Bitte verändert diese Datei nicht !!! Schreibt stattdessen in eine Datei `.zshrc_local` – falls sie nicht vorhanden ist, legt sie an und schreibt sie in euer Homeverzeichnis. Sie wird dann automatisch beim Öffnen einer neuen Shell berücksichtigt. Falls ihr die `Bash` verwendet, habt ihr ganz analog eine Datei namens `.bashrc` und solltet im Homeverzeichnis eine Datei namens `.bashrc_local` anlegen – falls sie nicht schon vorhanden ist.

Die Variable $PERL5LIB kann dann praktischerweise auch beim Installieren neuer Module verwendet werden.

CPAN-Module installieren

(a) manuell

Herunterladen des Moduls von `www.cpan.org` (am besten direkt auf `www.cpan.org/modules` gehen, oder über `www.perl.com/CPAN/`); dann

```
tar xvzf Some_Module_4.54
cd Some_Module_4.54
perl Makefile.PL  LIB=$PERL5LIB # falls $PERL5LIB nicht gesetzt ist: LIB=~/perl5
make
make test
make install
```

Oft erfährt man dann aber bei der Installation eines Moduls, dass dieses Modul wiederum das Vorhandensein anderer Module voraussetzt, die aber noch nicht installiert sind. Also muss man erst diese anderen Module installieren, die sich aber dann eventuell wiederum über fehlende Module beschweren. Dieser Prozess kann sehr lästig werden.

(b) mit der CPAN-Shell

Mit der CPAN-Shell wird die Installation stark vereinfacht. Zum Start wird einfach der Befehl `cpan` aufgerufen. Beim erstmaligen Start wird automatisch eine Konfigurationsdatei angelegt. Dann erscheint der Prompt der CPAN-Shell und man kann mit dem Befehl `install <Modul>` (wobei `<Modul>` der Modulname ist) ein Modul installieren. Dabei werden alle dafür benötigten anderen Module automatisch mitinstalliert. Mit dem Befehl `quit` wird die Shell beendet. Die CPAN-Shell installiert die Module per Default im Verzeichnis `~/perl5/`. Die Umgebungsvariable `$PERL5LIB` sollte also entsprechend gesetzt werden.

Eine Übersicht über die möglichen Kommandos erhält man mit `help`. Verlassen kann man die CPAN-Shell mit `exit`. Informationen über installierte und verfügbare Versionen eines Moduls erhält man mit `m <Modulname>`.

Wenn ich ein Modul installiert habe, wo finde ich dann die Dokumentation dazu?

Einfach auf der Shell `perldoc Some::Module` aufrufen.

11.3 Objektorientierte Module

Wie viele andere Programmiersprachen (bspw. C++) bietet auch Perl die Möglichkeit der objektorientierten Programmierung. Objekte fassen Daten und Funktionen, die auf diesen Daten operieren, zu einer Einheit zusammen. Implementiert werden Objekte mit Hilfe von objektorientierten Modulen.

Klassen und Objekte

Objekte gehören immer einer Klasse an und sind Instanzen davon. Ein Beispiel: In einem Bibliotheksverwaltungsprogramm könnte man die Klasse „Buch" definieren und als Instanzen davon (also als einzelne Objekte) jeweils die Bücher. In einer Univerwaltung könnte man allgemein die Klasse „Student" definieren, und die einzelnen Studenten wären programmtechnisch dann als Objekte dieser Klasse definiert.

Würde man das Ganze in PERL programmieren, würde man ein Modul namens „Student" schreiben – d.h. normalerweise fallen dann Modul-, Package- und Klassenname zusammen. In dieser Klasse (= diesem Modul) wären dann mehrere Variablen definiert (z.B. Name, Vorname, Alter, Studienfach) sowie Subroutinen, z.B. zum Abfragen oder Neubelegen der Variablen. Diese Variablen und Subroutinen gehören einerseits zur Klasse, können aber dann auch bezogen auf jedes einzelne Objekt aufgerufen werden. In Verbindung mit Klassen und Objekten nennt man die Subroutinen allerdings nicht Subroutinen, sondern „Methoden".

Arbeiten mit Objekten

Programmtechnisch gesehen arbeitet man eigentlich nie mit den Objekten direkt, sondern immer mit Referenzen auf Objekte. Dementsprechend erhält man üblicherweise eine Referenz zurück, wenn man ein neues Objekt einer Klasse erzeugt. Dies ist auch der Grund dafür, dass man praktisch immer die Pfeil-Notation verwendet, um Variablen oder Methoden einer Klasse bzw. eines Objekts anzusprechen. Ein paar Beispiele:

```
1  $student_1 = Student->new();
```

würde ein neues Objekt der Klasse „Student" erzeugen und eine Referenz darauf in `$student_1` ablegen. Diese Anweisung würde nicht in dem Klassenmodul selbst stehen, sondern dieses nur benutzen.

Wenn in der Klasse die Methoden `set_name()` und `set_first_name()` definiert sind, dann könnte man so weiterschreiben:

```
1 $student_1->set_name("Schmidt");
2 $student_1->set_first_name("Harald");
```

Ist auch die Methode `get_name()` in der Klasse definiert, dann könnte man den Namen wie folgt abfragen:

```
1 $name = $student_1->get_name();
```

Während man Objekte immer erzeugen muss, braucht man sie normalerweise nicht explizit zu zerstören – die Referenzen sind Variablen, und wenn ihr Gültigkeitsbereich verlassen wird, werden sie von PERL intern gelöscht, ohne dass man sich darum kümmern muss.

Schreiben einer Klasse

Im Prinzip macht man nichts weiter, als das Package (= den Namen der Klasse) zu setzen und dann die Methoden zu definieren.

Allerdings muss man Folgendes wissen:

1. Jeder Methode wird automatisch ein Argument übergeben, selbst wenn man die Methode später ohne Argumente aufruft. Wir haben zwei Arten von Aufrufen gesehen:

   (a) `$student_1 = Student->new();`
   Hier wird der Klassenname benutzt, um die Methode `new()` aufzurufen, nicht ein Objekt. („Student" ist ja kein Objekt, sondern die Klasse selbst.)

   (b) `$student_1->set_name("Schmidt");`
   Hier wird die Methode über ein konkretes Objekt (`$student_1`) aufgerufen.

   Wird die Methode über den Klassennamen aufgerufen, so wird als erstes Argument an die Methode `new` automatisch ein String übergeben, der (normalerweise) den Klassennamen enthält. Im zweiten Fall (Aufruf über ein Objekt) wird der angesprochenen Methode als erstes Argument eine Referenz auf dieses Objekt übergeben. Erst nach diesem automatisch übergebenen Argument folgen dann in der Argumentliste (`@_`) die explizit beim Aufruf genannten Argumente.

2. Variablen, die zu einem Objekt gehören (um z.B. Name, Vorname, Geburtstag, Studienfach, etc. zu speichern) werden normalerweise intern in Form eines Hashes verwaltet. Das ist nicht zwingend so, ist aber aus praktischen Gründen die verbreitetste Form; auf andere wollen wir hier nicht eingehen.

3. Um festzulegen, dass ein Objekt einer bestimmten Klasse angehören soll, benutzt man in PERL den Befehl `bless()`. Erstes Argument ist das Objekt, zweites die Klasse (oder, falls kein zweites Argument angegeben wird, das aktuelle Package).

Der Konstruktor

Mit diesen Angaben haben wir alle Informationen, um den typischen Aufbau der Methode `new` zu verstehen. Diese Methode, die neue Objekte der betreffenden Klasse erzeugt, nennt man Konstruktor. Üblicherweise gibt man der Konstruktor-Methode den Namen `new()`, auch wenn das nicht zwingend ist.

Hier ein typischer Konstruktor:

```
1  sub new {
2      my $class = shift;
3  # bei Aufruf ueber den Klassennamen ist das
4  # erste Argument der Klassenname
5      my $self = {};
6  # das ist schon unser Objekt in Rohversion;
7  # eine Referenz auf ein leeres Hash!
8      bless($self,$class);
9  # damit wird aus unserer Referenz wirklich ein Objekt, das auch
10 # alle Methoden zur Verfuegung hat die in der Klasse definiert sind
11     return $self;
12 # jetzt brauchen wir nur noch unser Objekt zurueckzugeben
13 # (d.h. genauer gesagt eine Referenz darauf.
14
15 }
```

Ein Beispiel für ein objektorientiertes Modul

```
1  package Student;
2  use strict;
3
4  sub new {
5          my $class = shift;
6          my $self = {};
7          bless($self, $class);
8          return $self;
9  }
10
11
12 sub set_name {
13         my $self = shift;
14         $self->{NAME} = shift;
15 }
16
17 sub get_name {
18         my $self = shift;
19         return $self->{NAME};
20 }
21
22 1; # nicht vergessen, es handelt sich ja um ein Modul
```

Diese Klasse könnte man z.B. wie folgt benutzen:

```
1  use strict;
2  use warnings;
3
4  use Student;
5
6  my $test_student = Student->new();
7  $test_student->set_name("John Smith");
8  my $name = $test_student->get_name();
9  print "Name des Studenten: $name\n";
10
11 # Output: "Name des Studenten: John Smith"
```

Ein paar Verbesserungen

Vor dem Befehl `bless()` könnte man dann auch schon einen Teil einbauen, um Variablen zu initialisieren (in diesem Fall mit `undef`):

```
1 $self->{NAME} = undef;
2 $self->{VORNAME} = undef;
3 ...
```

Außerdem kann man auch einplanen, dass die Methode `new()` nicht über den Klassenname (d.h. als sog. Klassenmethode), sondern über ein schon existierendes Objekt (d.h. als sog. Objektmethode) aufgerufen wird. Dann ist das erste Argument in der Argumentliste nicht der Klassennamen, sondern eine Referenz auf ein Objekt.

Was wir aber in jedem Fall für unseren `bless`-Befehl haben wollen, ist der Klassennamen. Falls wir nun ein Objekt (genauer gesagt eine Referenz auf ein Objekt) haben, liefert uns der Befehl `ref` den Namen der Klasse, zu der das Objekt gehört:

```
1 $object = shift;
2 $class = ref($object);
```

Da wir allerdings nicht von vornherein wissen, ob wir als erstes Argument den Klassennamen erhalten oder eine Objektreferenz, können wir uns wie folgt behelfen:

```
1 $arg = shift;
2 $class = (ref($arg) or $arg);
```

Haben wir als erstes Argument eine Objektreferenz erhalten, wird `ref($arg)` ausgewertet und liefert den Klassennamen. Da dieser logisch gesehen wahr ist, wird der zweite Teil der or-Verknüpfung nicht mehr ausgewertet. Ergibt `ref($arg)` jedoch keinen wahren Wert (weil wir doch einen Klassennamen als erstes Argument erhalten haben), dann wird `$arg` ausgewertet. Womit wir auch in diesem Fall den Klassennamen haben.

Eine weitere übliche Praxis besteht darin, nicht 2 getrennte Methoden `get_name()` und `set_name()` zu schreiben. Stattdessen schreibt man eine einzige Methode. Wird sie mit einem Argument aufgerufen, so belegt sie die betreffende Variable mit diesem Wert und gibt ihn auch als Rückgabewert zurück. Wird sie ohne Argument aufgerufen, gibt sie nur den Wert der Variablen zurück, so wie er vorher schon war:

```
1 sub name {
2     my $self = shift;
3     if(@_) {
4         $self->{NAME} = shift;
5     }
6     return $self -> {NAME};
7 }
```

Somit könnte dann eine etwas verbesserte Version unserer Klasse „Student" wie folgt aussehen:

```
1  package Student;
2
3  use strict;
4  use warnings;
5
6  sub new {
7      my $arg = shift;
8      my $class = (ref($arg) or $arg);
9      my $self = {};
10     $self->{NAME} = undef;
11     $self->{AGE} = undef;
12     bless($self, $class);
13     return $self;
14 }
15
16 sub name {
17     my $self = shift;
18     if(@_) {
19         $self->{NAME} = shift;
20     }
21     return $self -> {NAME};
22 }
23
24 sub age {
25     my $self = shift;
26     if(@_) {
27         $self->{AGE} = shift;
28     }
29     return $self->{AGE};
30 }
31
32 1;
```

Diese Version könnte man dann mit folgendem Programm benutzen:

```
1  use strict;
2  use warnings;
3
4  use Student;
5
6  my $test_student = Student->new();
7  $test_student->age(20);
8  $test_student->name("John Smith");
9  my $age = $test_student->age();
10 my $name = $test_student->name();
11 print "$name: $age\n";
12
13 # Output: "John Smith: 20"
```

11.4 Übungsaufgaben

Hinweis zu den Aufgaben 9.1 und 9.2: Modul und Testprogramm sollten immer im selben Verzeichnis stehen.

Aufgabe 11.1 Kopiere `My_module_2bfinished.perl` in dein Arbeitsverzeichnis und wandele es in ein funktionierendes Modul namens `StripHTML.pm` um. Das Modul soll auch eine Versionsangabe enthalten. Die Funktion `strip_html` soll defaultmäßig exportiert werden, es soll aber auch möglich sein, `%entity2char` zu importieren.

Aufgabe 11.2 Schreibe ein kleines Test-Programm, das das Modul benutzt, um aus dem Test-HTML-String "<html>Dies ist eine
Übung</html>" die HTML-Tags zu entfernen; außerdem soll die Versionsnummer des Moduls ausgegeben werden (ohne sie zu importieren).

Aufgabe 11.3 Schreibe ein Mini-Programm, das das Modul `CPAN` aufruft (ohne es weiter zu benutzen) und folgende Information ausgibt:

(a) Welche Version hat das Modul?

(b) Welche Variablen werden *per default* in das aufrufende Programm importiert?
Hinweis: Die Variablen, die innerhalb des Moduls die gesuchte Information enthalten, müssen über den Package-Namen angesprochen werden.

Aufgabe 11.4 Installiere das Modul `Lingua::EN::Inflect` mit Hilfe der CPAN-Shell. Dann schreibe ein kleines Testprogramm, dass mit Hilfe des Moduls den Plural des Wortes "goose" ausgibt (Hinweis: `PL` muss explizit importiert werden).

Aufgabe 11.5 Schreibe ein kleines Programm, das das Modul `LWP::Simple` einbindet (das Modul sollte bereits überall installiert sein) und dann folgendes tut:

- den Benutzer nach einer Webseite fragen
- den HTML-Code dieser Webseite in eine Variable einlesen (daran denken, dass zu einer vollständigen URL auch das `http://` am Anfang gehört)

Und, wenn das Einlesen erfolgreich war:

- Ausgabe des Textes ohne HTML auf den Bildschirm

Andernfalls:

- Ausgabe einer Fehlermeldung

Aufgabe 11.6 Schreibe ein Modul, in dem eine Objektklasse vom Typ `Book` definiert wird. Titel, Autor und Jahr müssen schon beim Aufruf des Konstruktors als Argumente übergeben werden, und im Objekt werden entsprechende Variablen mit diesen Werten belegt. Mit Hilfe von drei Methoden soll die Klasse das Abfragen (nicht das Setzen!) von Titel, Autor bzw. Jahr erlauben. Schreibe zu dieser Klasse ein kleines Testprogramm, das 2 Objekte dieser Klasse schafft, und dann deren Daten abfragt und auf dem Bildschirm ausgibt.

12

Agenten und Roboter

12.1 Agents / Webrobots

Programme, die für einen Benutzer im Netz bestimmte Aufgaben erledigen (z.B. eine Software herunterladen, eine bestimmte Information suchen, oder auch aus Mailing-Listen interessante Mails herausfiltern und dem Benutzer schicken), nennt man Agents. Auch Webbrowser können zu den Agents gezählt werden; das Wort wird auf sehr unterschiedliche Programmtypen angewendet.

Programme, die selbständig und meistens auch über längere Zeit im Internet navigieren, nennt man Robots oder kurz Bots. Eine typische Art von Robots sind z.B. Crawler, die Seiten herunterladen, daraus die Links extrahieren, und die so gefundenen Seiten wieder herunterladen etc. (Diese Technik verwenden z.B. die meisten Websuchmaschinen). Auch dieser Begriff wird jedoch nicht immer einheitlich verwendet.

Da solche Robot-Programme in sehr hoher Frequenz Anfragen an ein und denselben Server generieren und abschicken können, kann die Belastung eines Servers durch Robot-Programme sehr hoch werden. Wenn die Belastung zu hoch wird, kann dies auch zum kompletten Absturz des Servers führen. Viele Webadministratoren versuchen daher, den Zugriff von Robots auf ihre Server einzuschränken.

Ein anderer häufiger Grund für das „Aussperren" von Robots ist, dass bestimmte Bereiche zwar für Einzelanfragen abrufbar sein sollen, das Herunterladen von Daten in ihrer Gesamtheit aber unerwünscht ist.

12.2 robots.txt

`robots.txt` ist eine Datei, in die Webadministratoren hineinschreiben, welche Bereiche eines Servers für Robots abfragbar sein sollen und welche nicht. Sie ist, wenn vorhanden, unter `<Server-Adresse>/robots.txt` abrufbar.

Beispiel: `http://www.google.com/robots.txt` (diese Liste wächst stetig weiter an):

```
User-agent: *
Allow: /searchhistory/
Disallow: /news?output=xhtml&
Allow: /news?output=xhtml
Disallow: /search
Disallow: /groups
Disallow: /images
Disallow: /catalogs
Disallow: /catalogues
Disallow: /news
Disallow: /nwshp
Allow: /news?btcid=
Disallow: /news?btcid=*&
Allow: /news?btaid=
Disallow: /news?btaid=*&
Disallow: /setnewsprefs?
Disallow: /index.html?
Disallow: /?
Disallow: /addurl/image?
Disallow: /pagead/
Disallow: /relpage/
Disallow: /relcontent
Disallow: /sorry/
Disallow: /imgres
Disallow: /keyword/
Disallow: /u/
Disallow: /univ/
Disallow: /cobrand
Disallow: /custom
Disallow: /advanced_group_search
Disallow: /googlesite
Disallow: /preferences
Disallow: /setprefs
Disallow: /swr
Disallow: /url
Disallow: /default
Disallow: /m?
Disallow: /m/?
Disallow: /m/ig
Disallow: /m/lcb
Disallow: /m/news?
Disallow: /m/setnewsprefs?
Disallow: /m/search?
Disallow: /m/trends
Disallow: /wml?
Disallow: /wml/?
Disallow: /wml/search?
Disallow: /xhtml?
```

```
Disallow: /xhtml/?
Disallow: /xhtml/search?
Disallow: /xml?
Disallow: /imode?
Disallow: /imode/?
Disallow: /imode/search?
Disallow: /jsky?
Disallow: /jsky/?
Disallow: /jsky/search?
Disallow: /pda?
Disallow: /pda/?
Disallow: /pda/search?
Disallow: /sprint_xhtml
Disallow: /sprint_wml
Disallow: /pqa
Disallow: /palm
Disallow: /gwt/
Disallow: /purchases
Disallow: /hws
Disallow: /bsd?
Disallow: /linux?
Disallow: /mac?
Disallow: /microsoft?
Disallow: /unclesam?
Disallow: /answers/search?q=
Disallow: /local?
Disallow: /local_url
Disallow: /froogle?
Disallow: /products?
Disallow: /froogle_
Disallow: /product_
Disallow: /products_
Disallow: /print
Disallow: /books
Disallow: /patents?
Disallow: /scholar?
Disallow: /complete
Disallow: /sponsoredlinks
Disallow: /videosearch?
Disallow: /videopreview?
Disallow: /videoprograminfo?
Disallow: /maps?
Disallow: /mapstt?
Disallow: /mapslt?
Disallow: /maps/stk/
Disallow: /maps/br?
Disallow: /mapabcpoi?
Disallow: /center
Disallow: /ie?
Disallow: /sms/demo?
Disallow: /katrina?
Disallow: /blogsearch?
Disallow: /blogsearch/
Disallow: /blogsearch_feeds
Disallow: /advanced_blog_search
Disallow: /reader/
Disallow: /uds/
Disallow: /chart?
Disallow: /transit?
Disallow: /mbd?
Disallow: /extern_js/
Disallow: /calendar/feeds/
Disallow: /calendar/ical/
Disallow: /cl2/feeds/
Disallow: /cl2/ical/
Disallow: /coop/directory
Disallow: /coop/manage
Disallow: /trends?
Disallow: /trends/music?
Disallow: /notebook/search?
Disallow: /music
Disallow: /musica
Disallow: /musicad
Disallow: /musicas
Disallow: /musicl
Disallow: /musics
Disallow: /musicsearch
Disallow: /musicsp
Disallow: /musiclp
Disallow: /browsersync
Disallow: /call
Disallow: /archivesearch?
Disallow: /archivesearch/url
Disallow: /archivesearch/advanced_search
Disallow: /base/search?
Disallow: /base/reportbadoffer
Disallow: /movies?
Disallow: /codesearch?
Disallow: /codesearch/feeds/search?
Disallow: /wapsearch?
Disallow: /safebrowsing
Disallow: /reviews/search?
Disallow: /views?
```

Eine Datei kann eine oder mehrere Absätze "User Agent" + zugehörige "Disallow"-Felder enthalten (oft ist es wie im Beispiel nur ein Absatz). Unter "User-Agent" ist eingetragen, auf welche Robots sich der Absatz bezieht. $*$ bedeutet dabei: alle User Agents, die in keinem anderen Absatz angesprochen werden. (Der Beispiel-Absatz bezieht sich also auf alle Robots überhaupt, da es keine anderen Absätze gibt.) Unter Disallow stehen dann die Verzeichnisse, auf die die angesprochene Art von Robots nicht zugreifen darf.

Da bei Google die Suchergebnisse im Verzeichnis /search zurückgegeben werden, heißt das, dass Google überhaupt keine Abfrage seiner Suchergebnisse durch Robots wünscht.

Die Eintragungen in `robots.txt` bedeuten nicht, dass der Zugriff unmöglich wäre. Es handelt sich sozusagen um eine Aufforderung, diese Bereiche doch bitte nicht mit Robots abzufragen.

Um sich effektiv gegen Nichtbeachtung zu schützen, setzen viele Webadministratoren Programme ein, die die Zahl der Zugriffe überwachen und woher diese kommen. Kommen zu viele Anfragen in kurzer Zeit von ein und derselben Internet-Adresse, so werden Anfragen, die von dieser Adresse abgeschickt werden, nicht mehr bearbeitet, der Robot wird also effektiv ausgesperrt – und jegliche weiteren Anfragen von dieser Adresse aus, d.h. dann auch manuelle Aufrufe. Das heißt, dass man riskiert, von einer Seite dauerhaft ausgesperrt zu werden, falls man robots.txt nicht beachtet. Dann kann man nur noch versuchen, durch eine entschuldigende Mail an den Webmaster der betroffenen Site eine Wiederfreischaltung zu erlangen.

12.3 LWP::UserAgent

Das PERL-Modul, das es uns erlaubt, sehr bequem Webseiten herunterzuladen heißt `LWP::UserAgent`. Im Unterschied zu `LWP::Simple` gibt es uns sehr viel mehr Informationen über den Ablauf unserer Anfragen und erlaubt eine genauere Steuerung. `LWP::UserAgent` nimmt uns jedoch nicht die Arbeit ab, auf `robots.txt` zu achten und / oder auf Wartezeiten, bevor derselbe Server erneut kontaktiert wird.

Bei der Verwendung von `LWP::UserAgent` kommen normalerweise immer auch zwei weitere Objektklassen zum Einsatz: `HTTP::Request` und `HTTP::Response`. Nachdem man einen User Agent, d.h. ein

Objekt vom Typ `LWP::UserAgent` erzeugt hat (und gewisse Parameter gesetzt hat), besteht das eigentliche Herunterladen von Seiten darin, dass man pro Anfrage ein Objekt der Klasse `HTTP::Request` erzeugt und dieses der Methode `request()` des User Agent übergibt. Der Rückgabewert dieser Methode ist dann ein Objekt vom Typ `HTTP::Response`, das u.a. den Inhalt der angefragten Webseite enthält.

Ein Beispiel für die Nutzung von LWP::UserAgent

```
1  use LWP::UserAgent;
2
3  # neues UserAgent-Objekt erstellen
4  $my_ua = LWP::UserAgent->new();
5
6  # Eigenschaften bestimmen
7
8  $my_ua->agent('libwww-perl/6.05');   # Das ist der Default
9  $my_ua->from('stiehler@cis.uni-muenchen.de');
10 $my_ua->timeout(60);
11 $my_ua->max_size(307200); # max 300kB
12
13 ###################################
14 # File holen und lokal speichern #
15 ###################################
16
17 $my_ua->mirror('http://www.cis.uni-muenchen.de', 'test.html');
18
19 #####################################
20 # File holen zur Weiterverarbeitung #
21 #####################################
22
23 # neues Request-Objekt erstellen
24 $my_request = HTTP::Request->new('GET', 'http://www.cis.uni-
      muenchen.de');
25
26 # interessante Methoden von HTTP::Request
27 $req_string = $my_request->as_string();
28 print STDERR "Request: $req_string";
29
30 # Request an User-Agent uebergeben
31 # Dieser erzeugt ein HTTP::Response-Objekt
32 $my_response = $my_ua->request($my_request);
33
34 # interessante Methoden von HTTP::Response
35 $resp_status = $my_response->status_line();
36 print STDERR "$resp_status\n";
37
38 if ($my_response->is_success) {
39   print $my_response->content;
40 } else {
41   print $my_response->error_as_HTML; # wird sowieso auf STDERR
        ausgegeben
42 }
43
44 # Beispielanwendung:
45 open(OUT, ">down.html");
46 print STDERR "Please enter a URL to download (or <return> to end):
      ";
47
```

```
48 while(<>) {
49   chomp;
50   last unless $_;
51   $my_request = HTTP::Request->new('GET', $_);
52   $my_response = $my_ua->request($my_request);
53   if ($my_response->is_success) {
54     $req_for_this = $my_response->request()->as_string();
55     $base_for_this = $my_response->base();
56     print STDERR "Request: $req_for_this\n";
57     print STDERR "Base-URL: $base_for_this\n";
58     print OUT "Request: $req_for_this\n";
59     print OUT "Base-URL: $base_for_this\n";
60     print OUT $my_response->content;
61     print STDERR "Page has been stored\n";
62   } else {
63     print OUT $my_response->error_as_HTML;
64   }
65   print STDERR "Please enter a URL to download (or <return> to end)
       : ";
66 }
67
68 close OUT;
```

Ein Punkt, den man dabei im Auge haben sollte sind Redirects. Redirects bedeuten, dass in manchen Webdokumenten *Umleitungen* eingebaut sind. Man erhält nicht den Inhalt der angefragten Webseite, sondern den Inhalt der in der Umleitung angegebenen Webseite. Beispiel: Man gibt `www.mercedes.de` in einem Browser ein und landet ohne eigenes Zutun auf `www.mercedes-benz.de`. Dies bedeutet,

(a) dass in solchen Fällen nicht nur eine Abfrage abgeschickt wird, sondern als Reaktion auf das erste *Redirect* noch eine zweite zu der neuen Adresse (und in manchen Fällen kann sich das auch noch weiter fortsetzen).

(b) dass das Response-Objekt, das wir erhalten, nicht unbedingt auf unseren ursprünglichen Request hin erfolgt ist. Deshalb kann

```
1 $my_response->request()->as_string()
```

durchaus etwas anderes enthalten als

```
1 $my_request->as_string()
```

(c) dass auch die Basis-URL nicht mehr unbedingt die von der ursprünglichen Anfrage ist. Will man z.B. relative Links in der Webseite wieder rekonstruieren, sollte man sich unbedingt vorher die Basis-URL noch einmal aus dem Response-Objekt holen:

```
1 $base_url = $response->base;
```

Es gibt übrigens bei Response-Objekten auch eine Methode, die den Zugriff auf das jeweilige vorhergehende Response-Objekt ermöglicht, falls mehr als eine Anfrage erfolgte.

12.4 URI::Heuristic

Das Modul stellt uns 2 Funktionen (=Subroutinen) zur Verfügung:

```
1 $expanded_url = uf_uristr($raw_url);
2 $my_uri_object = uf_uri($raw_url);
```

Das Modul ist NICHT objektorientiert. D.h. um die Funktionen nutzen zu können, braucht kein Objekt geschaffen zu werden. Das Modul wird einfach mit `use` eingebunden, die gewünschten Funktionen werden importiert und stehen dann wie eine selbst definierte Subroutine zur Verfügung.

Beide Funktionen expandieren Strings mittels Heuristiken in komplette URLs. `uf_uristr()` gibt einen String zurück, `uf_uri` eine Referenz auf ein URI-Objekt. `uf` in den Methoden-Namen soll übrigens für `user friendly` stehen. Ob das sehr benutzerfreundlich ist, den Methoden solche Namen zu geben? Hier ein paar Beispiele für die Umwandlungen:

```
perl                -> http://www.perl.com
www.oreilly.com     -> http://www.oreilly.com
ftp.funet.fi        -> ftp://ftp.funet.fi
/etc/passwd         -> file:/etc/passwd
```

Als Länder-Code wird defaultmäßig derjenige des Hostrechners genommen, in Deutschland würde z.B. aus "perl" `http://www.perl.de`, nicht `http://www.perl.com`.

12.5 LWP::RobotUA – Automatische Beachtung der Netiquette

Beachtung der „Netiquette" (der Verhaltensregeln im Internet) bedeutet für Robots zweierlei:

1. Beachtung von `robots.txt`
2. Vermeidung von hochfrequenten Anfragen an ein und denselben Server.

Der zweite Punkt wird automatisch beachtet, wenn man einen User Agent nicht mit `LWP::UserAgent` sondern mit `LWP::RobotUA` generiert.

Die Methoden sind weitgehend identisch, bis auf folgende (ausgewählte) Unterschiede:

new:

```
1 $rua = LWP::RobotUA->new($agent_name, $from, [$rules])
```

Der Konstruktor muss obligatorisch mit einem Agenten-Namen und einer E-Mail-Adresse aufgerufen werden.

Als 3. Argument kann optional ein `WWW::RobotsRules-Objekt` übergeben werden. Diese Art von Objekten sorgt intern für die Beachtung der `robots.txt`-Einträge. Gibt man kein 3. Argument an, legt sich der `RobotUA` selbst ein neues Objekt dieser Klasse an, was normalerweise das gewünschte Verhalten ist.

delay:

Die `delay`-Methode stellt ein, wie lange der `RobotUA` warten soll, ehe er denselben Server ein zweites Mal kontaktiert. Der Wert setzt die Wartezeit in Minuten. Default ist 1, d.h. eine Minute Wartezeit. Es können nicht nur ganze Zahlen als Argument benutzt werden:

```
1 $rua->delay(0.5);
```

stellt jeweils eine zeitliche Distanz von 30 Sekunden zwischen 2 Anfragen an denselben Server sicher.

Achtung: `RobotUA` wartet auch vor der ersten Anfrage an einen vorher noch nie kontaktieren Server.

12.6 HTML::LinkExtor

Der `HTML::LinkExtor` ist eine Klasse zum Extrahieren von Links aus Webpages. Er ist eine Subklasse von `HTML::Parser`.

Subklassen:

Wenn man als Programmierer eine neue Klasse erstellen will, dann bietet es sich oft an, auf schon bestehende Klassen zurückzugreifen. Man sagt: meine Klasse soll zunächst einmal genau die Eigenschaften der existierenden Klasse haben. Dann ändert man Methoden und / oder fügt neue hinzu – und fertig ist die gewünschte eigene Klasse. Das Ganze nennt man in der objektorientierten Programmierung *Vererbung*.

HTML::LinkExtor als Subklasse von HTML::Parser

Das heißt:

(a) `HTML::LinkExtor` ist ein HTML-Parser, der außerdem noch ein paar zusätzliche Methoden zum Extrahieren von Links hat.

(b) In den Man-Pages zum `HTML::LinkExtor` sind nur die Methoden beschrieben, in denen sich `HTML::LinkExtor` vom `HTML::Parser` unterscheidet. Die Dokumentation für die anderen Methoden muss man in den Man-Pages zum `HTML::Parser` nachschlagen.

Methoden von HTML::LinkExtor

parse_file()

nimmt als Argument eine Datei in Form eines Strings, eines Filehandles oder einer Referenz auf ein Filehandle:

```
1  $my_extractor->parse_file("cis_homepage.html");
```

Der Inhalt der Datei wird geparst. Rückgabewert ist im Erfolgsfall eine Referenz auf das Objekt selbst, mit dem man die Methode aufgerufen hat, oder – im Falle eines Fehlers – `undef` (die Fehlermeldung steht dann in $!). Der Rückgabewert ist also nur für die Kontrolle interessant, ob die Datei eingelesen werden konnte. Die eigentlichen Ergebnisse können mit der Methode `links` abgefragt werden.

links

Wurde ein HTML-Text geparst, dann können mit der Methode `links` die Links abgerufen werden:

```
1  @links = $my_extractor->links;
```

Die Links werden zurückgegeben als *Array of Arrays*, wobei jedes der einzelnen Arrays als erstes Element den Link-Typ enthält, gefolgt von Paaren Attribut-Name – Attribut-Wert.

Z.B. würde der HTML-Ausschnitt

```
<A HREF="http://www.thomas-fahle.de">Thomas Fahle</A>
<IMG SRC="xyz.gif" LOWSRC="low_xyz.gif"></IMG>
```

ein Array mit folgenden Elementen zurückliefern:

Element 0: eine Referenz auf das Array ('a' , 'href', 'http://www.thomas-fahle.de'),
Element 1: eine Referenz auf das Array ('img', 'src', 'xyz.gif', 'lowsrc', 'low_xyz.gif').

Wird `links` abgefragt, so wird danach intern das *Array of Arrays* gelöscht. `links` kann also nur einmal pro geparstem Dokument abgefragt werden.

Beispiel 1: Extrahieren aller Links aus einer abgespeicherten Webseite:

```
 1  #!/usr/bin/perl
 2
 3  use strict;
 4  use warnings;
 5
 6  use HTML::LinkExtor;
 7
 8  my $html_file = shift;
 9  my $p = HTML::LinkExtor->new();
10
11  $p->parse_file($html_file)
12     or die "$!";
13
14  foreach my $link ($p->links()) {
15    print "@{$link}\n";
16  }
```

Output von Beispiel 1:

```
parker~/<2>webmodule> extract_links.perl cis_homepage.html
link href /styles/cis_neu.css
a href http://www.uni-muenchen.de/
a href /CISengl.html
img src /images/eng.gif
img src /images/usflag.gif
a href /CISLage.html
a href CISStudienberatung/CISStudiengaenge.html
a href CISStudienberatung
a href /events/CISEventsAlt.html
a href CISProjekte.html
a href CISPublikationen.html
a href CISMitarbeiter.html
a href studenten/CISStudenten.html
a href /studenten/fachschaft/
a href /CISSIL.html
a href CISBibliothek.html
a href System
a href /links/links.html#computerlinguistik
a href /links/clin.html
a href /links/links.html#uni
a href /links/links.html#bibliotheken
a href /links/links.html#info
a href /CISengl.html
a href MAILTO:webmaster@cis.uni-muenchen.de
```

Probleme:

1. Man will Webseiten oft nicht erst speichern, bevor sie man parsen kann.

2. Die Links sind relativ, normalerweise will man absolute Links (d.h. mit dem kompletten Pfad)

Zunächst zu den absoluten Pfaden: Hier helfen uns die Argumente, die wir dem Konstruktor eines `HTML::LinkExtor`-Objekts mitgeben können.

new:

Der Konstruktor kann mit keinem, einem oder zwei Argumente aufgerufen werden.

Aufruf ohne Argumente:

```
1 $my_extractor = HTML::LinkExtor->new();
```

Optionales Argument 1:

Eine Subroutine, die jeweils im Anschluss an die Methode `parse` aufgerufen wird (eine sog. *callback*-Routine, die werden wir im Moment nicht verwenden. Sie ist nur interessant, wenn man den Input in kleinen Portionen, sog. *Chunks*, verarbeiten will, sobald er über das Netz hereingekommen ist.)

```
1 $my_extractor = HTML::LinkExtor->new(\&callback);
```

Optionales Argument 2:

Eine URL, die allen relativen Links vorangestellt wird, um sie zu komplettieren:

```
1 $my_extractor =
2 HTML::LinkExtor->new(\&callback, 'http://www.cis.uni-muenchen.de');
```

Wird z.B. in der angegebenen Webseite ein Link auf `/CISengl.html` gefunden, kommt es bei diesem Aufruf von `new` von `links` zurückgegeben als

`http://www.cis.uni-muenchen.de/CISengl.html`

Will man dieses zweite optionale Argument (die sog. *base url*) setzen, ohne das erste zu benutzen, dann schreibt man anstelle des ersten Arguments `undef`:

```
1 $my_extractor =
2 HTML::LinkExtor->new(undef, 'http://www.cis.uni-muenchen.de');
```

Beispiel 2:

```
1  #!/usr/bin/perl
2
3  use strict;
4  use warnings;
5
6  use HTML::LinkExtor;
7
8  my $html_file = shift;
9  my $base_url = shift;
10
11 my $p = HTML::LinkExtor->new(undef, $base_url);
12 $p->parse_file($html_file) or die "$!";
13
14 foreach my $linkinfo ($p->links() ) {
15   print "@{$linkinfo}\n";
16 }
```

Output von Beispiel 2:

```
parker~/<2>webmodule>
extract_links_with_base.perl cis_homepage.html http://www.cis.uni-muenchen.de

link href http://www.cis.uni-muenchen.de/styles/cis_neu.css
a href http://www.uni-muenchen.de/
a href http://www.cis.uni-muenchen.de/CISengl.html
img src http://www.cis.uni-muenchen.de/images/eng.gif
img src http://www.cis.uni-muenchen.de/images/usflag.gif
a href http://www.cis.uni-muenchen.de/CISLage.html
a href http://www.cis.uni-muenchen.de/CISStudienberatung/CISStudiengaenge.html
a href http://www.cis.uni-muenchen.de/CISStudienberatung
a href http://www.cis.uni-muenchen.de/events/CISEventsAlt.html
a href http://www.cis.uni-muenchen.de/CISProjekte.html
a href http://www.cis.uni-muenchen.de/CISPublikationen.html
a href http://www.cis.uni-muenchen.de/CISMitarbeiter.html
a href http://www.cis.uni-muenchen.de/studenten/CISStudenten.html
a href http://www.cis.uni-muenchen.de/studenten/fachschaft/
a href http://www.cis.uni-muenchen.de/CISSIL.html
a href http://www.cis.uni-muenchen.de/CISBibliothek.html
a href http://www.cis.uni-muenchen.de/System
a href http://www.cis.uni-muenchen.de/links/links.html#computerlinguistik
a href http://www.cis.uni-muenchen.de/links/clin.html
a href http://www.cis.uni-muenchen.de/links/links.html#uni
a href http://www.cis.uni-muenchen.de/links/links.html#bibliotheken
a href http://www.cis.uni-muenchen.de/links/links.html#info
a href http://www.cis.uni-muenchen.de/CISengl.html
a href MAILTO:webmaster@cis.uni-muenchen.de
```

Aber: Es ist ziemlich umständlich, jedes Mal zusätzlich zu einer Datei auch noch die Basis-URL anzugeben. Und sowieso wollen wir die Dokumente ja direkt parsen, wenn wir sie aus dem Web holen.

12.7 Webseiten aus dem Netz holen und die Links extrahieren

Um die Webseiten aus dem Netz zu holen und ohne Zwischenspeichern zu verarbeiten, bietet sich natürlich `LWP::RobotUA` an.

parse:

Die geeignete Methode von `HTML::LinkExtor`, um die von `LWP::RobotUA` geholten Seiten ohne vorheriges Abspeichern zu bearbeiten, ist dann natürlich nicht mehr `parse_file`, sondern `parse`. Sie nimmt einen HTML-Text als Argument, so wie er von der `content`-Methode des `LWP::RobotUA` zurückgegeben wird. Das eigentliche Ergebnis, d.h. die Links, stehen wie bei `parse_file` nicht im Rückgabewert, sondern sie können mit der Methode `links` abgerufen werden.

Redirects

Um *Redirects* einzukalkulieren, sollte man auf jeden Fall aus dem Response-Objekt die Basis-URL abfragen, die beim letzten *Response* tatsächlich ermittelt wurde, und nicht die Basis-URL, mit der man seine Anfrage gestartet hat:

```
1 $base_url = $response->base;
2 # so bekommt man die korrekte Basis-URL der
3 # tatsaechlich heruntergeladenen Webseite
```

Mit dieser Basis-URL kann jeweils ein neues LinkExtor-Objekt geschaffen werden, und die relativen Links werden dann automatisch in absolute Links umgewandelt.

12.8 Übungsaufgaben

Aufgabe 12.1 Schreibe ein Programm, das in einer Schleife

1. den Benutzer nach einer Webadresse fragt,
2. die Eingabe mit Hilfe von URI::Heuristic expandiert,
3. den Inhalt der Webadresse ins Programm lädt
4. die abgeschickte Anfrage (= Request) ausgibt,
5. diejenige Anfrage ausgibt, die letztlich zur gefundenen Seite geführt hat,
6. die Base-URL und den Titel der Seite ausgibt bzw. die entsprechende Fehlermeldung, falls die Anfrage nicht funktioniert hat. Die Methode, um den Titel auszugeben heißt einfach `title()`.

Gib in das Programm eine Seite mit *Redirect* ein (z.B. www.yahoo.de) und vergleiche die abgeschickte Anfrage mit derjenigen, die dann tatsächlich zum Herunterladen der Seite geführt hat.

Aufgabe 12.2 Schreibe einen Crawler, der als Argument eine Startseite empfängt, die die Hauptseite einer nicht allzu umfangreichen DMOZ-Kategorie ist, z.B. `http://www.dmoz.org/World/Deutsch/Wissenschaft/Ingenieurwissenschaften/Luft-_und_Raumfahrt/Raumfahrt/`.

Der Crawler soll

(a) die Kategorie aus dem Pfad der Startseite extrahieren (Für die oben genannten Startseiten wären das z.B. "Raumfahrt",),

(b) dann mittels LWP::RobotUA (der auf eine halbe Minute Wartezeit zwischen den Anfragen eingestellt ist!!!) diese Startseite holen und mit HTML::LinkExtor die Links aus ihr extrahieren,

(c) dann die gefundenen Links analysieren und ggfs. die Webseiten herunterladen, zu denen die Links führen:

- führt das Link zu einer externen Webseite mit Inhalt, der die gewünschte Kategorie betrifft, so soll die Webseite heruntergeladen und abgespeichert werden (alle Webseiten in eine einzige Datei; getrennt durch eine Trennlinie, gefolgt von einer Zeile mit der URL, danach dann der HTML-Code der Seite).
- führt das Link zu einer weiteren Katalog-Seite, und enthält der Pfad zu dieser Katalog-Seite immer noch die gewünschte Kategorie, so soll diese Seite heruntergeladen und ebenso bearbeitet werden wie die Startseite.
- andere Links sollen ausgefiltert werden (und dann nicht weiter bearbeitet werden); ebenso Katalogseiten, in deren URL das gesuchte Thema nicht mehr auftritt.

Dieser Vorgang soll fortgesetzt werden, bis alle Webseiten heruntergeladen wurden, zu denen die externen Links aus allen Katalog-Webseiten der vorgegebenen Kategorie führen. (Dies soll das Programm zumindest prinzipiell können. Es kann auch eine Abbruchbedingung eingebaut werden, die den Prozess nach Bearbeitung von 5 Katalogseiten abbricht).

Um besser die Unterscheidung treffen zu können, wie die einzelnen Links in Schritt c) einzuordnen sind, empfiehlt es sich, in einem ersten Schritt zunächst alle Links der Startseite auf dem Bildschirm auszugeben, so wie HTML::LinkExtor sie zurückgibt. Alle Links, die nicht mit „a href" anfangen, können dabei schon aussortiert werden. Links, die „dmoz" enthalten, sollten genau dann aussortiert

werden, wenn es nicht weitere Katalogseiten zum Thema sind (d.h. bei weiteren Katalogseiten zum Thema muss das Thema im Pfad der URL auftreten). Hat man diese Vorsortierung durchgeführt, kann es sein, dass es noch eine Art von Links gibt, die weder auf weitere Katalogseiten zum Thema noch auf externe Links zum Thema verweisen. Diese Art von Links sollte auch noch herausgefiltert werden. Bei allen anderen Links kann dann davon ausgegangen werden, dass es sich um externe Links zum Thema der Katalogseite handelt.

Wer will, kann einen vorgefertigten Programmanfang benutzen.

Wer den vorgefertigten Programmanfang benutzt und die Ausgabe der Links schon darin eingebaut hat, braucht dann eigentlich nur noch folgende Schritte durchzuführen:

- Die Liste der extrahierten externen Katalogseiten muss an die aufrufende Routine zurückgegeben werden.
- In einer Subroutine sollte eine URL aus dem Netz geladen und das Response-Objekt zurückgegeben werden (soweit nicht schon für das Ausgeben der Links geschehen).
- In einer weiteren Subroutine (die die eben genannte Subroutine zum Herunterladen aufruft) sollten die externen Links zum Thema geholt und an die Ausgabedatei angehängt werden (mit einer Kontrolle, ob dieses Link nicht schon bearbeitet worden ist).
- Auch eine Subroutine, die aus einem Response-Objekt die Links extrahiert und zurückgibt, hilft, die einzelnen Subroutinen kurz zu halten.

Bei der Subroutine, die URLs aus dem Netz herunterlädt, empfiehlt es sich, das RobotUA-Objekt nur einmal zu erzeugen und mittels Reference Counting am Leben zu erhalten, z.B.:

```
1  BEGIN {
2
3      my $link_ua = LWP::RobotUA->new('LWP::RobotUA','meine.
           mail@adresse.com');
4      $link_ua->timeout(30);
5      $link_ua->max_size(1000000); # oder was immer ihr als angemessen
           betrachtet
6      my $link_delay = 0.5;
7
8      # Hier koennen auch andere Variablen verwaltet werden, deren
9      # Werte nach Beenden der Subroutine erhalten bleiben sollen.
10
11     sub get_page {
12         ...
13     }
14 ...
15 }
```

Aber natürlich sind das alles nur Vorschläge, das Programm darf auch beliebig anders realisiert werden, solange die Aufgabenstellung und die allgemeinen Regeln zum Programmieren eingehalten werden (die Aufgabenstellung kann z.B. auch rekursiv gelöst werden).

Sonstige Hinweise:

- auch zum Testen nie einen Delay unter 0.2 benutzen!!! Alle anderen aus dem Kurs dürften auch gerade am Testen sein, und dann multiplizieren sich natürlich die Anfragen, die gleichzeitig von unseren IP-Adressen kommen.

- Bitte achtet darauf, den Gültigkeitsbereich der Variablen so sehr einzugrenzen wie möglich: Es sollen keine globalen Variablen benutzt werden!
- Auch ansonsten sind die Hinweise zum Schreiben von Programmen hier besonders zu beachten.
- Falls jemand die RobotUA-Methode `host_wait` nutzen will, um den Programmbenutzer über die noch verbleibende Wartezeit bis zur nächsten Server-Anfrage zu informieren (ist aber nicht obligatorisch, sondern nur eine Anregung für diejenigen, die noch etwas mehr lernen wollen): Als Argument darf nicht der Host-Name allein angegeben werden, sondern Host-Name + Portnummer, d.h. am Ende des Host-Namens muss i.d.R. noch ein „:80" angefügt werden, um das gewünschte Ergebnis zu erhalten.

Aufgabe 12.3 Erweitere das Programm so, dass der Delay jeweils auf 0 eingestellt wird, wenn ein vorher noch nie kontaktierter Server angefragt werden soll, und danach wieder auf den ursprünglich gewählten Delay-Wert zurückgesetzt wird. Der Server wird über den Hostnamen identifiziert, das ist der URL-Teil zwischen '://' und, soweit vorhanden, dem ersten '/'.

13

Sockets

Mit Hilfe von Sockets können Programme Daten untereinander in beiden Richtungen austauschen. Dabei gibt es meist ein Programm, das Anfragen von anderen Programmen entgegennimmt und sie beantwortet (ein sog. Server) und beliebig viele andere Programme, die ihre Anfragen an den Server schicken und mit den Antworten weiterarbeiten (die sog. Clients). Das ganze System ist auch bekannt als Client-Server-Programmierung.

Ein typisches Beispiel sind Webserver: der Webserver wartet auf Anfragen von Browsern (oder *User Agents* o.ä.) und gibt Webseiten zurück. Die Browser arbeiten dann mit diesen Webseiten weiter, indem sie sie anzeigen und Operationen darauf erlauben (Anzeigen des Quelltextes etc.).

13.1 Seinen Kommunikationspartner finden: IPs und Ports

Eine IP-Adresse ist eine Adresse im Internet, die aus 4 Nummern besteht, von denen jede im Bereich von 0 bis 255 liegt. Jeder Rechner, der ans Internet angeschlossen ist, hat eine solche IP-Adresse und kann über sie eindeutig identifiziert werden. Unser Rechner `diener` ist z.B. als `129.187.148.24` zu erreichen.

Wenn ich in einem Browser eine Seite wie z.B. `http://www.cis.uni-muenchen.de/` aufrufe, wird diese intern zuerst in die entsprechende IP-Adresse übersetzt, um überhaupt mit dem entfernten Rechner Kontakt aufnehmen zu können. Diese „Übersetzung“ übernimmt rechnerintern ein Programm namens `DNS`. Kann intern einer Webadresse keine IP-Adresse zugeordnet werden, dann kann der andere Rechner nicht angesprochen und logischerweise die Seite auch nicht abgerufen werden.

Um mit einem bestimmten Programm auf einem anderen Rechner zu kommunizieren, reicht es allerdings nicht, einfach den anderen Rechner als solchen anzusprechen – er weiß sonst ja nicht, welches Programm auf ihm die Anfrage bearbeiten soll. Deshalb muss jedes Server-Programm sich zuallererst eine Port-Nummer sichern, unter der es zu erreichen ist, und jeder Client muss dem entfernten Rechner diese Port-Nummer mitteilen, damit der die Anfrage dem richtigen Programm zuleiten kann. Dabei gibt es einige Port-Nummern, die normalerweise für bestimmte Programme vorreserviert sind. Eine Anfrage an ein Webserver-Programm wird z.B. defaultmäßig an Port 80 geleitet. Statt `http://www.cis.uni-muenchen.de/` kann ich daher genauso gut explizit Port 80 ansprechen (`http://www.cis.uni-muenchen.de:80/`), das Ergebnis ist dasselbe. Mit `http://www.cis.uni-muenchen.de:81/` hingegen wird meine Browser-Anfrage scheitern.

Programmiert man selbst Sockets, so sollte man natürlich vermeiden, eine solche reservierte Port-Nummer belegen zu wollen. Üblicherweise wählt man für eigene Ports Nummern größer als 1024, um Konflikte zu vermeiden, da die Standarddienste kleineren Port-Nummern zugeordnet sind.

13.2 Verbreitete Arten von Sockets

(a) Streams

(b) Datagrams

Streams verbrauchen mehr Systemressourcen als Datagrams, aber dafür ist nur mit ihnen eine verlässliche bidirektionale Kommunikation möglich. Datagrams werden wir hier nicht weiter behandeln.

13.3 Domains, in denen Sockets funktionieren

(a) Unix/Linux

(b) Internet

Unix-Domain-Sockets erlauben nur die Kommunikation von Programmen, die beide auf demselben Unix-Rechner laufen. Wir werden hier nur Internet-Sockets behandeln, die über verschiedene Rechner hinweg kommunizieren können.

13.4 Protokolle

Damit die Programme erfolgreich miteinander kommunizieren können, muss der Ablauf der Kommunikation natürlich vorab festgelegt sein. Eine solche Vereinbarung über den Ablauf der Kommunikation nennt man ein Protokoll. Ein Browser und ein Webserver z.B. können auf Basis unterschiedlicher Protokolle kommunizieren – HTTP (HyperText Transfer Protocol), FTP (File Transfer Protocol) o.a. Ist das Protokoll definiert, wissen beide Seiten genau, wie ihre Anfragen auszusehen haben bzw. (vom Server aus gesehen), wie das genau zu interpretieren ist, was vom Client da hereinkommt. Wir werden uns nur sehr "von außen"mit diesen Protokollen beschäftigen, aber man sollte sich vor Augen halten, dass die wirkliche Kommunikation auf Systemebene sehr viel mehr Detail-Vereinbarungen benötigt als wir bei PERL auf Programmierebene mitbekommen.

Wir werden uns hier nur mit Internet-Stream-Sockets beschäftigen, und da gibt es nicht viel auszuwählen. Das dafür verwendete Protokoll nennt sich TCP.

13.5 Das PERL-Modul IO::Socket::INET

Mit `IO::Socket::INET` kann man sehr leicht Sockets programmieren. Für die Einbindung reicht `use IO::Socket`. Die Generierung eines konkreten Sockets geschieht über den Konstruktor (d.h. die `new`-Methode) von `IO::Socket::INET`, dem man mehrere Argumente mitgibt. Die mitgegebenen Argumente entscheiden darüber, ob es ein Server- oder ein Client-Socket wird.

Beispiel: Ein Client-seitiges Socket

```
1    my $socket = IO::Socket::INET->new(PeerAddr => $remote_host,
2                                       PeerPort => $remote_port,
3                                       Proto    => "tcp",
4                                       Type     => SOCK_STREAM)
5      or die "Couldn't connect to $remote_host:$remote_port : $@\n";
```

Beispiel: Ein Server-seitiges Socket

```
$server = IO::Socket::INET->new(LocalPort => $server_port,
                                Proto    => "tcp",
                                Type      => SOCK_STREAM,
                                Reuse     => 1,
                                Listen    => SOMAXCONN )
    or die "Couldn't be a tcp server on port $server_port : $@\n";
```

Beim Server-Socket braucht man keine Rechneradresse angeben. Das ist ja automatisch die, auf der das Socket läuft. Nur einen Port muss man sich aussuchen. Wird das Server-Socket nicht ordnungsgemäß geschlossen, so kann normalerweise derselbe Port nicht sofort wiederbenutzt werden. Dies kann man allerdings durch Setzen von `Reuse` umgehen, was also insbesondere für die Phase der Programmentwicklung äußerst nützlich ist, wenn man den Server öfter abbricht.

`Listen` gibt an, wieviele Anfragen maximal in eine Warteschleife aufgenommen werden sollen. Ist diese Warteschleife voll, so wird die Anfrage nicht bearbeitet, und der Client erhält eine entsprechende Meldung. `SOMAXCONN` ist eine Systemkonstante die angibt, wie viele Anfragen das System maximal in der Warteschleife zulässt. Sie hat z.B. den Wert 128.

Der Client kann `$socket` jetzt wie ein Filehandle benutzen, d.h. hineinschreiben, daraus lesen und es schließen, mit denselben Befehlen, die man auch für Filehandles benutzt.

Hier ein Beispiel für ein komplettes Client-Programm, das insgesamt 10 Sockets einrichtet, in sie hineinschreibt, sie dann ausliest und wieder schließt:

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

use IO::Socket;
use strict;

my $remote_host = "pc39.cis.uni-muenchen.de";
my $remote_port = 1789;

foreach my $i (1..10) {
  my $socket = IO::Socket::INET->new(PeerAddr => $remote_host,
                                     PeerPort => $remote_port,
                                     Proto    => "tcp",
                                     Type     => SOCK_STREAM)
    or die "Couldn't connect to $remote_host:$remote_port : $@\n";

  print STDERR "Asking:\tHow much is $i * $i?\n";
  print $socket "How much is $i * $i?\n";
  my $answer = <$socket>;
  print STDERR "Receiving:\t$answer\n";
  close($socket);
}
```

Der Server wartet gewöhnlich in einer Endlos-Schleife auf hereinkommende Anfragen. Die Methode dafür ist `accept()`. Solange keine Anfrage hereinkommt, bleibt das Programm beim `accept()`-Aufruf stehen. Kommt dann eine Anfrage, gibt `accept()` als Rückgabewert die neue Verbindung zum anfragenden Client zurück und die Anfrage kann bearbeitet werden.

Beispiel:

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

use IO::Socket;
use strict;

my $server_port = 1789;

my $server = IO::Socket::INET->new(LocalPort => $server_port,
                                   Type      => SOCK_STREAM,
                                   Reuse     => 1,
                                   Listen    => SOMAXCONN )
    or die "Couldn't be a tcp server on port $server_port : $@\n";

my $answer;
while (my $client = $server->accept()) {
  # $client is the new connection

  my $question = <$client>;
  chomp($question);
  print STDERR "Receiving:\t$question\n";

  if ($question =~ /is\s*(\d+\s*[+\-*\/]\s*\d+)\?/) {
     $answer = eval($1);
  } else {
    $answer = "Sorry, I didn't understand your request";
  }

  print STDERR "Answering:\t$answer\n";
  print $client "$answer\n";

  close($client);
  print STDERR "\nWaiting for the next connection ...\n\n";

}

close($server);
```

In diesem Fall wird aus Anschaulichkeitsgründen eine einfache, wenn auch wenig effiziente Absprache zwischen Client und Server zugrundegelegt: pro Verbindung eine Anfrage und eine Antwort, danach schließt jede Seite die Verbindung wieder. In anderen Fällen dürfte es praktischer sein, dieselbe Verbindung für mehrere Anfragen zu nutzen, aber dann muss dem Server auch jeweils signalisiert werden, wann er die Verbindung auf seiner Seite schließen soll – entscheidend ist auf jeden Fall, dass auf beiden Seiten dieselben Regeln gelten und nicht der eine versucht zu schreiben oder zu lesen, während der andere schon die Verbindung zugemacht hat.

Beide Programme basieren übrigens auch darauf, dass ihnen bekannt ist, dass die zu erwartende Nachricht nur aus einer Zeile besteht. Sonst müsste man wie bei einem Filehandle auch eine Schleife über <> laufen lassen.

13.6 Pufferung

Bevor irgendwelche Daten effektiv an File-Handles geschrieben werden, werden sie systemintern gewöhnlich gepuffert. D.h. es wird gewartet, bis eine bestimmte Menge Daten zusammengekommen ist, und erst dann werden die Daten tatsächlich übertragen. Bei Sockets ist dies gewöhnlich so lange kein Problem, wie Zeilen übertragen werden, denn normalerweise werden solche Puffer jeweils

bei einem Zeilenumbruch rausgeschrieben. Sollen Zeichen oder Zeichenketten ohne "\n" übertragen werden, muss man dafür sorgen, dass die Puffer geleert werden, sobald etwas in sie hineingeschrieben wird. Dies kann man erreichen, indem man zunächst mit dem Befehl `select()` das betreffende Handle/Socket zum aktuellen *Default*-Handle macht (auf das alle Ausgaben geleitet werden, z.B. von `print`-Befehlen, falls nicht explizit etwas anderes angegeben ist). Rückgabewert des `select`-Befehls ist dasjenige Handle, das vorher das *Default*-Handle war; dieses speichert man in einer Variablen (im Normalfall ist `main::STDOUT` das *Default*-Handle). Dann setzt man die Systemvariablen $| auf 1. Damit schaltet man für das *Default*-Handle (in diesem Fall das, das wir gerade mit select ausgewählt haben) auf „Sofortausgabe". Danach sollte man wieder das vorher „aktuelle" Handle mit `select()` auswählen:

```
1 $vorher_aktuelles_handle = select ($client);
2
3 $| = 1;
4
5 select ($vorher_aktuelles_handle);
```

Dies ist das allgemeine Vorgehen, das für alle Handles benutzt werden kann. Bei IO::Select-Objekten gibt es auch die Methode `autoflush()`, also z.B. `$client->autoflush()`. Ab Version 1.18 des IO::Select-Moduls ist `autoflush()` *per default* aktiviert.

Näheres zum Thema Pufferung siehe unter „Suffering from Buffering"[3].

13.7 Seinen Kommunikationspartner identifizieren

Der Client muss den Rechner und Port des Servers kennen, ehe er überhaupt mit ihm Kontakt aufnehmen kann. Der Server hingegen weiß nicht automatisch, von wo eine Anfrage kommt. Er kann dies jedoch folgendermaßen feststellen:

1. Rückgabewert von `$server->accept()` ist im skalaren Kontext nur das Client-Objekt. Im Listenkontext wird jedoch als 2. Wert die Client-Adresse mitübergeben. Die `accept()`-Zeile könnte also so lauten:

   ```
   1 while (my ($client, $client_address) = $server->accept()) {
   ```

2. `$client_address` enthält dann die Client-Adresse und den Port – aber in einem nicht direkt lesbaren Format kodiert. Mit der PERL-Funktion `sockaddr_in()` kann eine erste Umwandlung vorgenommen werden:

   ```
   1 my ($port, $packed_ip) = sockaddr_in($client_address);
   ```

   Das `in` am Ende von `sockaddr_in()` steht übrigens für Internet, da es sich um die Adresse eines Internet-Sockets handelt. Nach der Operation ist der Port dann lesbar, und die IP-Adresse steht jetzt auch in einer einzelnen Variablen; sie ist aber immer noch nicht direkt lesbar.

3. Die Einzelinformationen aus der gepackten IP-Adresse lassen sich mittels der PERL-Funktion `gethostbyaddr()` gewinnen:

   ```
   1 my ($name, $aliases, $addrtype, $length, @addr) = gethostbyaddr
       ($packed_ip, AF_INET);
   ```

   bzw. nur

   ```
   1 my $name = gethostbyaddr($packed_ip, AF_INET);
   ```

   wenn einen die anderen Informationen nicht interessieren.

[3] `http://www.foo.be/docs/tpj/issues/vol3_3/tpj0303-0002.html`

Damit hat man in `$name` den Namen des Client-Rechners und in `$port` den Port.

Hier nochmal als Programm-Ausschnitt:

```
1 while (my ($client, $client_address) = $server->accept()) {
2   # $client is the new connection
3   my ($port, $packed_ip) = sockaddr_in($client_address);
4   my $name = gethostbyaddr($packed_ip, AF_INET);
5   print "\nReceived a connection from $name, Port $port\n";
6
7   my $question = <$client>;
8   ...
9 }
```

Diese Methode ist nicht zuverlässig, falls der Name des anderen Rechners manipuliert wurde! Für eine sichere Lösung siehe Kap. 17.7 des PERL-Cookbooks.

13.8 Effizienter Kommunizieren: IO::Select

Nicht immer ist jeder ankommende Client schon bereit zum Auslesen. Dies kann daran liegen, dass der Client zwar die Verbindung schon aufgebaut hat, zwischenzeitlich aber andere Operationen durchführt und dann erst schreibt, oder es kann auch an der Pufferung liegen – dder Client schreibt, aber die Information wird noch nicht losgeschickt, weil der Puffer noch nicht voll ist.

Wenn der Server ein Socket auszulesen versucht, das noch nicht auslesefertig ist, wird wertvolle Zeit für die Abarbeitung anderer Clients verschenkt – und falls ein Client wegen eines Fehlers oder einer Manipulation überhaupt nicht auslesefertig wird, ist der Server komplett blockiert. So wie wir den Server bisher programmiert haben, arbeitet er ja immer einen Client nach dem anderen ab.

Das Modul `IO::Select` bietet durch seine `can_read`-Methode die Möglichkeit, alle gerade auslesefertigen Clients festzustellen, mit denen schon eine Verbindung besteht und auch solche Clients, die gerade eine Verbindung zum Server aufbauen wollen. Die auslesefertigen Clients werden dann in einem Array zurückgegeben – und für die auf eine Verbindung wartenden (für die kann ja noch keine Client-Verbindung zurückgegeben werden) das Server-Socket.

Beispiel:

```
1  #!/usr/bin/perl -w
2
3  use strict;
4  use IO::Select;
5  use IO::Socket;
6
7  my $server = new IO::Socket::INET(Listen => 1, LocalPort => 8080);
8  my $selector = new IO::Select( $server);
9
10 while(my @ready = $selector->can_read) {
11
12   foreach my $handle (@ready) {
13
14     if($handle == $server) {
15     # neue Client-Verbindung wartet darauf, akzeptiert zu werden
16
17       my $new_connection = $server->accept;
18       # kann normalerweise nicht blockieren, denn wir wissen ja,
             dass tatsaechlich ein Client auf die Verbindung wartet
```

```
19          $selector->add($new_connection);
20          # wir uebergeben dem Select-Objekt die neue Verbindung zur
                Ueberwachung
21
22        } else {
23
24          # eine bereits bestehende Verbindung ist auslesefertig
25
26          my $request = <$handle>;
27
28          # kann normalerweise auch nicht blockieren
29
30          if ($request) {
31
32            # Anfrage kann abgearbeitet werden; je nachdem danach
                  Schliessen der Verbindung
33
34          } else {
35
36            # der Client hat mittlerweile die Verbindung seinerseits
                  geschlossen
37
38            $selector->remove($handle);
39            # das Selector-Objekt soll die Verbindung nicht laenger
                  ueberwachen
40
41            $handle->close;
42            # wir schliessen unsererseits ebenfalls die Verbindung
43
44          }
45        }
46      }
47  }
```

Zuerst wird das Server-Socket generiert. Danach das Select-Objekt. Dieses führt intern stets eine Liste der aktuellen Verbindungen – initialisiert wird diese Liste mit dem Server-Socket. Dann wird als Start einer `while`-Schleife überprüft, welche Sockets auslesefertig sind bzw. für das Server-Socket, ob eine Anfrage an den Server existiert. Beim ersten Durchlauf kann nur das Server-Socket zurückgegeben werden. Dies eröffnet eine neue Verbindung zu dem wartenden Client. Dieser wird nicht direkt bearbeitet (da er vielleicht ja noch gar nicht auslesefertig ist), sondern nur vom Select-Objekt in seine Liste der aktuellen Verbindungen aufgenommen. Bei allen weiteren Durchläufen wird wieder eine Liste aller auslesefertigen Sockets / wartender Verbindungsanfragen bearbeitet: das Server-Socket akzeptiert, falls ein Client sich verbinden will, eine neue Verbindung, alle anderen auslesefertigen Client-Verbindungen werden abgearbeitet oder gelöscht und aus der Liste entfernt, falls doch nichts mehr von ihnen ankommt.

Eine andere Möglichkeit der effizienten Abarbeitung ist es, für jede neue Client-Verbindung einen eigenen Subprozess zur Abarbeitung zu starten.

13.9 Übungsaufgaben

Aufgabe 13.1 Schreibe ein Client- und ein Server-Socket, die miteinander kommunizieren.

Aufgabe 13.2 Ändere das Programm aus Aufgabe 13.1 so ab, dass der Server ausgibt, von welchem Rechner aus er kontaktiert wurde.

Aufgabe 13.3

(a) Schreibe ein Server-Programm (mit `IO::Select`), das eine Liste von URLs erhält und auf jede Client-Anfrage hin jeweils die nächste URL aus der Liste als Antwort zurückschickt.

(b) Schreibe dazu passend ein Client-Programm, das sich in einer Schleife jeweils eine URL vom Server holt, diese auf dem Bildschirm ausgibt und dann 5 Sekunden nichts tut (mit dem `sleep`-Befehl). Der Server soll so lange URLs an die Clients verteilen, bis keine mehr in seiner Liste sind. Starte mehrere Instanzen des Clients parallel; jeder sollte verschiedene URLs ausgeben, aber insgesamt sollten alle URLs ausgegeben werden; jeder Client soll so lange arbeiten, bis er vom Server keine URL mehr bekommt.

Hinweise zur Aufgabe 13.3:

- Die Schleifenbedingung im Server könnte dann z.B. so aussehen:

```
1 while (@urls and (my @ready = $selector->can_read)) {
2   ...
3 }
```

  Wenn keine URLs mehr zu vergeben sind, empfiehlt es sich, keine Nachrichten mehr an die Clients zu schicken, sondern einfach serverseitig alle Verbindungen zuzumachen. Nach dem Ende der Schleife sollten also auch noch die evtl. verbliebenen Handles mit `$selector->handles()` ausgelesen und geschlossen werden. Der Client stellt dann jeweils fest, dass er keine Antwort mehr bekommt und kann daraus schließen, dass keine URLs mehr zu vergeben sind.

- Die einzelnen Client-Instanzen können sich bei Meldungen auf dem Bildschirm z.B. mit ihrer jeweiligen Prozess-Nummer (PID) melden; diese steht immer in $$.

- Man kann den Server und alle Client-Instanzen auf ein und derselben Shell starten, wenn man am Ende des Aufrufs jeweils ein & anfügt (= Starten im Hintergrund).

14

Aus 1 mach 2: Forking

Neben den Systemaufrufen (`system`, `exec`, Backticks und Start von Programmen mittels `open`) gibt es noch eine andere, oft benutzte Möglichkeit, einen neuen Prozess aus PERL heraus zu starten: mittels `fork`. Damit ruft man an einer beliebigen Stelle im Programm einen zweiten Systemprozess ins Leben, den so genannten Child-Prozess. Den ursprünglichen Prozess nennt man in diesem Zusammenhang Parent-Prozess. Beide Prozesse bearbeiten den nach der `fork`-Anweisung folgenden Code.

Normalerweise will man aber nicht, dass derselbe Programmcode auf einmal von zwei Prozessen abgearbeitet wird, sondern man will, dass der eine Prozess einen für ihn bestimmten Teil des Programms abarbeitet und der andere Prozess einen anderen Teil. Für diese Aufteilung muss man selber sorgen. Wie aber stellt man fest, von welchem Prozess der aktuelle Code bearbeitet wird? Die Lösung liegt im Rückgabewert des `fork`-Befehls – genauer gesagt, in den beiden unterschiedlichen Rückgabewerten. Aus Sicht des Parent-Prozesses gibt `fork` eine positive Zahl zurück, nämlich die Prozess-ID des Child-Prozesses. Aus Sicht des neuen Child-Prozesses gibt `fork` hingegen Null zurück. Man braucht also nur den Rückgabewert von `fork` zu speichern und dann mittels `if`-Bedingungen abzufragen (bitte noch nicht ausprobieren!!!):

```
1 $child_pid = fork;
2
3 if ($child_pid) {
4     # alles was hier steht, wird jetzt vom Parent-Prozess
          abgearbeitet, denn aus seiner Sicht ist die Bedingung wahr
5     # der Child-Prozess ueberspringt diesen Code-Teil, da aus
          seiner Sicht die Bedingung nicht wahr ist
6 } else {
7     # alles was hier steht, wird jetzt vom Child-Prozess
          abgearbeitet
8     # der Parent-Prozess ueberspringt diesen Code-Teil, da aus
          seiner Sicht schon die if-Bedingung erfuellt war
9 }
```

In dem Moment, wo sich das Programm durch eine `if-else`-Anweisung mit einem `fork` in der `if`-Bedingung in zwei Prozesse aufteilt, werden alle Variablen mit ihren Inhalten kopiert – ab dieser Stelle hat also jeder der beiden Prozesse alle bisher benutzten Variablen mit ihren aktuellen Belegungen zur Verfügung; Änderungen dieser Belegungen wirken sich aber nur noch innerhalb des eigenen Prozesses aus, der andere Prozess bekommt davon nichts mit. Nur File-Handels werden nicht kopiert – beide Prozesse schreiben und lesen aus den- bzw. in dieselben Filehandles.

14.1 Vorsicht, Zombies!

Beenden sollte man auf jeden Fall immer erst den Child-Prozess, und der Parent-Prozess sollte mittels des `wait`-Befehls darauf warten. Andernfalls schafft man sog. Zombie-Prozesse, die weiterhin auf dem Rechner verwaltet werden, obwohl sie nichts mehr machen. Das sollte vermieden werden. (Das Betriebssystem hält terminierte Child-Prozesse weiter in der sog. *process table*, damit ihr Fehlerstatus noch abgefragt werden kann. `wait` tut genau dies, so dass der terminierte Child-Prozess dann aus der Prozessverwaltung herausgenommen wird).

Eine andere Sache, auf die man sehr achten sollte, ist, den Child-Prozess immer mit der `exit`-Anweisung zu beenden. Andernfalls arbeitet der Child-Prozess weiter und führt ggfs. Programmcode aus, der eigentlich dem Parent-Prozess zugeordnet ist.

14.2 Fehlerabfrage beim Forken

`fork` gibt normalerweise die Prozess-ID des Child-Prozesses zurück und man speichert diesen Wert in der Variablen `$pid`. Diese ist dann im Parent-Prozess verfügbar. Aus Sicht des Child-Prozesses, dessen Programmcode mit der `else`-Anweisung beginnt, ist dieselbe Variable mit Null belegt.

Ist etwas schiefgelaufen, gibt `fork` den Wert `undef` zurück. Um zu kontrollieren, ob alles funktioniert hat beim `fork`-Aufruf, kann man folgenden Code verwenden:

```
1  # Prozess-ID, d.h. ein Wert ungleich Null, wenn alles glatt
       gelaufen ist
2  if ($child_pid = fork) {
3
4     # Parent-Prozess
5
6  } elsif (defined $child_pid) {
7
8     # Null, aber definiert aus Sicht des Child-Prozesses, wenn alles
          glatt gelaufen ist
9     # => der Child-Prozess arbeitet gerade die Bedingung ab
10    # Child-Prozess
11
12 } else {
13
14    # $child_pid war undefiniert, es muss ein Fehler aufgetreten sein
15
16    die "Cannot fork: $!\n";
17 }
```

Fehlermeldungen, die von den Child-Prozessen erzeugt werden, finden sich übrigens nicht in $!, wie andere Fehlermeldungen im Programm, sondern in $? (Fehler des Forkens selber werden aber in $! abgelegt, dafür ist ja noch kein Child-Prozess verantwortlich).

14.3 Ein erstes Beispiel

Anhand eines kleinen Beispiels soll die Funktionsweise von parallel arbeitenden Prozessen demonstriert werden. Das Programm soll sich aufteilen, sowohl Parent- als auch Child-Prozess sollen ein paar Berechnungen vornehmen, danach schließt der Parent-Prozess das Programm ab.

```
1  #!/usr/bin/perl
2
3  use strict;
4  use warnings;
5
6  my $basis = 3;
7  print "Die Basis ist $basis\n";
8  my $child_pid = 0;
9
10 # Prozess-ID, d.h. ein Wert ungleich Null, wenn alles glatt
       gelaufen ist
11
12 if ($child_pid = fork) {
13     sleep(1);
14     print "parent: basis = $basis\n";
15
16     for my $x (1..10) {
17         sleep(2);
18         print "parent: $basis hoch $x = ", $basis ** $x, "\n";
19     }
20
21     my $wait_return_value = wait;
22     # nicht vergessen, sonst gibt's einen Zombie-Prozess!
23
24     if ($wait_return_value != $child_pid) {
25         die "Der Child-Prozess wurde nicht richtig beendet!\n $?";
26     }
27
28 } # Null, aber definiert aus Sicht des Child-Prozesses, wenn alles
       glatt gelaufen ist
29
30   elsif (defined $child_pid) {
31     print "child: ich aendere jetzt die Basis!\n";
32     $basis++;
33     print "child: Die Basis ist fuer mich jetzt $basis\n";
34
35     for my $x (1..10) {
36         sleep(2);
37         print "child: $basis hoch $x = ", $basis ** $x, "\n";
38     }
39     exit;
40     # nicht vergessen, sonst arbeitet der Child-Prozess den
          folgenden Programmcode auch noch ab!
41
42 } # $child_pid war undefiniert, es muss ein Fehler aufgetreten sein
43
44   else {
45     die "Fehler beim Forken: $!\n";
46 }
47
48 print "Ok, Programm beendet!\n";
49 # nur der Parent-Prozess kommt bis hier
```

Während `$basis` für den Parent-Prozess mit dem ursprünglichen Wert belegt bleibt, ändert der Child-Prozess ihn ab – wovon der Wert der Variablen im Parent-Prozess aber nicht beeinflusst wird. `STDOUT` wird von beiden Prozessen gemeinsam genutzt – beide schreiben auf `STDOUT`, beide Ausgaben erscheinen auf dem Bildschirm.

Massenweise Kinderchen ...

Mit `fork` kann man aber auch aus einem Prozess mehr als zwei Prozesse machen – man forkt einfach mehrfach. Dabei forkt normalerweise immer der Parent-Prozess. Natürlich kann man auch einen Child-Prozess wieder forken lassen, aber das kompliziert die Überwachung der Beendigung der Child-Prozesse. Auf der Seite `http://www.steve.gb.com/perl/lesson12.html` gibt es ein gut kommentiertes Beispiel, wie man insgesamt 10 Child-Prozesse erzeugt.

... aber übertreiben sollte man es auch nicht

Man kann also rein von der Programmierung her beliebig viele Child-Prozesse ins Leben rufen. Allerdings sollte man sich darüber im Klaren sein, dass diese Child-Prozesse annähernd soviel System-Ressourcen beanspruchen wie ein normal laufendes PERL-Programm. Das heisst, es kommt annähernd dem elffachen parallelen Starten eines PERL-Programms nahe, wenn man innerhalb eines Perlprogramms 10 Child-Prozesse startet. Dementsprechend wird die CPU belastet – und andere Ressourcen, insbesondere die Netzlast, falls die Child-Prozesse viel über das interne Rechnernetz oder über das Internet kommunizieren.

14.4 waitpid()

Beim Abspalten von mehr als einem Child-Prozess sollte man nicht mehr mit `wait()` arbeiten, sondern mit `waitpid()`. Der Unterschied ist, dass `wait()` nur auf den zuletzt abgespaltenen Prozess wartet – das ist natürlich ungünstig, wenn man auf die Rückkehr von insgesamt 10 noch ausstehenden Child-Prozessen warten will. Bei `waitpid()` gibt man als Argument die Prozess-ID des Child-Prozesses an, und genau auf diesen Prozess wartet das Programm dann. Als weiteres Argument muss man noch einen Wert mitgeben, über den man noch besonderes Verhalten steuern kann – eine Null tut es für unsere Zwecke.

Rückgabewert ist, wenn alles funktioniert hat, die Prozess-ID des beendeten Child-Prozesses. Andernfalls wird -1 zurückgegeben. Mit Hilfe dieses Rückgabewertes kontrolliert auch das Programm mit den 10 Child-Prozessen, ob wirklich alle Child-Prozesse beendet sind oder irgendwo noch ein Zombie-Prozess sein Unwesen treibt.

14.5 Forking und Sockets

Interessante Möglichkeiten der Programmierung ergeben sich in der Kombination von Forking mit Sockets, und zwar in beide Richtungen. Zum einen wird Forking sehr oft innerhalb von Server-Sockets eingesetzt. Sobald eine neue Client-Anfrage hereinkommt, wird ein Child-Prozess abgespalten, der diese Anfrage bearbeitet – der Parent-Prozess ist also nicht mehr mit der Bearbeitung der eigentlichen Anfragen belastet, sondern nur noch mit Warten auf Anfragen und dem Abspalten jeweils eines Child-Prozesses.

Aber auch in der anderen Richtung kann man die Kombination gut nutzen. Man kann von einem Programmprozess mehrere Child-Prozesse abspalten, und dann im Parent-Prozess ein Server-Socket starten. Die Child-Prozesse machen jeweils einen Teil der Arbeit und starten zusätzlich jeder ein Client-Socket. Über die Socket-Kommunikation zwischen Parent- und Child-Prozessen kann dann die Arbeit gesteuert werden.

14.6 Übungsaufgaben

Aufgabe 14.1 Schreibe ein Programm, dass in einem Array 3 Webadressen enthält. Spalte dann 3 Child-Prozesse ab, von denen jeder den Inhalt einer Webadresse holt (der erste Child-Prozess die erste Webadresse, der zweite Child-Prozess die zweite, etc.) und jeweils in einer eigenen Datei abspeichert (nutze die Funktion getstore von `LWP::Simple`, ohne Fehlerkontrolle).

Tipp: Nutze eine Variable, in der vor dem Forken jeweils die Webadresse steht, die der neu entstehende Child-Prozess holen soll (der Child-Prozesss kopiert ja die Belegung der Variablen zum Zeitpunkt des Abspaltens). Überwache, dass die Child-Prozesse auch wirklich terminieren!

Aufgabe 14.2 Schreibe ein Programm, das 3 Child-Prozesse abspaltet, wobei jeweils die Prozess-ID des neu abgespaltenen Child-Prozesses auf dem Bildschirm ausgegeben wird. Idee des Programms ist, dass der Parent-Prozess die URLs an die Child-Prozesse verteilt und diese die empfangenen URLs herunterladen.

Der Parent-Prozess soll nach dem Abspalten der 3 Child-Prozesse zunächst eine Liste von URLs einlesen und ein Server-Socket starten (dabei den Parameter `Reuse` am besten auf 1 setzen), während die Child-Prozesse ein paar Sekunden warten (so dass der Parent-Prozess Zeit hat, das Server-Socket zu starten, 3 Sekunden sollten auf jeden Fall reichen), und dann jeweils ein Client-Socket starten (mit 127.0.0.1 als Remote Server – diese festgelegte Adresse ist ein Synonym für "local host").

Die Kommunikation zwischen Parent-Prozess und Child-Prozessen soll dann wie folgt ablaufen: Die Child-Prozesse senden jeweils ihre PID an das Server-Socket, und solange das Server-Socket noch URLs zu vergeben hat, antwortet es mit einer URL, deren Inhalt der Child-Prozess daraufhin holt und abspeichert – dann sendet das Client-Socket wieder seine PID, und so weiter, bis alle URLs (verteilt auf die verschiedenen Child-Prozesse) vergeben sind. Während dieses ganzen Prozesses soll der Parent-Prozess jeweils ausgeben, von welchem Child-Prozess eine Anfrage kommt und was er ihm antwortet, während die Child-Prozesse jeweils ausgeben, welche Webseite sie gerade holen.

Das Ende der Arbeit:
Hat das Server-Socket alle URLs vergeben, so antwortet es auf die Anfragen der Child-Prozesse statt mit einer URL mit dem String `"please die\n"`. Der Child-Prozess schließt daraufhin sein Client-Socket und führt daraufhin ein `exit(0)` durch, während der Parent-Prozess mit `waitpid()` überwacht, dass der Child-Prozess auch tatsächlich endet. Außerdem soll das Server-Socket für jedes `waitpid()` ausgeben, ob es auch tatsächlich Erfolg hatte.

Während ein Server-Socket meistens ewig läuft (d.h. bis es von außen abgebrochen wird), soll bei diesem Programm das Server-Socket nur solange laufen wie Child-Prozesse am Leben sind (also so was wie `while ($kidsalive){ ... }`). D.h. der Parent-Prozess muss diese Variable beim Abspalten eines Child-Prozesses jeweils heraufzählen und nach jedem erfolgreichen waitpid wieder herunterzählen. Sind keine Child-Prozesse mehr am Leben, endet die Server-Schleife, und das einzige, was der Parent-Prozess vor dem Programmende noch machen sollte, ist das Server-Socket zu schließen.

15

File Locking

15.1 Was ist File-Locking und wozu braucht man das?

In manchen Situationen greifen mehrere Prozesse gleichzeitig auf dieselbe Datei zu, was zu Problemen führen kann. Dies kann vorkommen, wenn beispielsweise ein CGI-Skript mehrfach aufgerufen wird und alle Instanzen des Programms auf dieselbe Datei zugreifen, oder wenn ein Programm durch Forking Kindprozesse erzeugt, die dieselbe Datei verwenden.

Solange alle Prozesse nur lesend zugreifen, ist die Situation unproblematisch. Erst wenn mindestens ein Prozess schreibend auf die Datei zugreifen will, können Probleme entstehen. Damit sich die verschiedenen Prozesse nicht in die Quere kommen und durch unkoordinierte Dateizugriffe Chaos entsteht, gibt es *File Locking* – ein System, mit dem mehrere Prozesse ihre Dateizugriffe koordinieren können.

15.2 Wie funktioniert File-Locking?

In PERL gibt es dafür den Befehl `flock(FILEHANDLE, FLAGS)`. Über die Flags lassen sich zwei verschiedene Arten des Zugriffs realisieren: gemeinsamer Zugriff („Shared Lock") mit dem Flag `LOCK_SH` und exklusiver Zugriff („Exclusive Lock") mit dem Flag `LOCK_EX`. Einen exklusiven Zugriff auf eine Datei würde man in PERL z.B. mit

```
1 flock(FH, LOCK_EX);
```

beantragen. Intern funktioniert das so: wenn ein Prozess ein „Shared Lock" verlangt, kontrolliert das Betriebssystem, ob ein anderer Prozess gerade einen „Exclusive Lock" auf die Datei ausübt. Wenn nicht, lässt das Betriebssystem den Prozess sofort zugreifen, andernfalls unterbricht es die Ausführung des anfragenden Prozesses, bis kein „Exclusive Lock" mehr auf die Datei ausgeübt wird. Erst dann kann der anfragende Prozess sein „Shared Lock" ausüben.

Ähnlich funktioniert die Anfrage nach einem „Exclusive Lock": Das Betriebssystem lässt den anfragenden Prozess warten, bis auf der betreffenden Datei von keinem anderen Prozess mehr ein „Shared Lock" oder ein „Exclusive Lock" ausgeübt wird, erst dann kann der anfragende Prozess zugreifen und übt sein „Exclusive Lock" aus.

Umgekehrt bedeutet das, dass bei Vorliegen eines „Exclusive Lock" alle anderen Prozesse warten müssen, egal ob sie einen „Shared Lock" oder einen „Exclusive Lock" beantragen. Bei Vorliegen eines „Shared Lock" müssen nur die Prozesse warten, die ein „Exclusive Lock" beantragen – aber diejenigen, die ein „Shared Lock" beantragen, können sofort zugreifen.

Da beim Schreiben in eine Datei Probleme für alle anderen Prozesse auftreten können, die gleichzeitig versuchen zu schreiben oder zu lesen, sollte man zum Schreiben immer ein „Exclusive Lock" beantragen – dann müssen alle anderen warten. Da gemeinsames Lesen aus einer Datei keine Probleme verursacht, reicht zum Lesezugriff ein „Shared Lock". So schließt man aus, dass gerade ein anderer

Prozess in die Datei schreibt.

Der Lock-Mechanismus funktioniert allerdings nur, wenn alle Prozesse vor ihren Zugriffen Locks beantragen. Er kann nicht verhindern, dass ein Prozess auf die Datei zugreift, ohne vorher ein Lock zu beantragen, egal ob gerade ein „Shared Lock" oder ein „Exclusive Lock" ausgeübt wird.

Aufheben eines Locks

Gibt man flock als Flag `LOCK_UN` mit, wird das Lock wieder aufgehoben. Schließt man ein Filehandle mit `close`, werden ebenfalls alle bestehenden Locks aufgehoben, ebenso am Programmende. Das Schließen des Filehandles zum Aufheben eines Locks ist dabei die sicherere Methode, denn nur dann werden automatisch alle Pufferinhalte noch übertragen, bevor das Lock aufgehoben wird.

Benutzung in PERL

Um die Flags nutzen zu können, muss man zunächst das Modul Fcntl einbinden, das die Konstanten der C-Header-Datei `fcntl.h` importiert. Am besten importiert man alles, was sich hinter den Tags `:DEFAULT` und `:flock` verbirgt. So hat man u.a. die Konstanten `LOCK_SH` und `LOCK_EX` zur Verfügung, die man als Flags beim Aufruf von `flock` verwenden kann.

15.3 Lesen aus einer Datei

Lesen ist die unproblematischste aller Operationen. Hier ist ein Beispiel:

```
1  use Fcntl qw(:DEFAULT :flock);
2
3  my $filename = "file.txt";
4  open(IN, "<$filename") or die "can't open $filename: $!";
5  flock(IN, LOCK_SH) or die "can't get a lock on $filename: $!";
6
7  while (<IN>) {
8    ...
9  }
10
11 close(IN) or die "$!"; # lock wird automatisch aufgehoben
```

15.4 Anhängen an eine Datei

Im Prinzip funktioniert das Anhängen genauso wie wie das Lesen: eine Datei zum Anhängen öffnen, Lock sichern (diesmal ein exklusives Lock), schreiben, und Datei wieder freigeben.

Das Problem ist: Was geschieht, wenn zwischen dem Moment des Öffnens der Datei und dem Erhalten des exklusiven Locks ein anderer Prozess etwas an die Datei anfügt? Dann gibt es trotz des File Lockings Chaos, weil unser Prozess nicht mitbekommen hat, dass sich das Dateiende verschoben hat. Er überschreibt dann, was der andere Prozess geschrieben hat. File Locking kann nur Probleme verhindern, nachdem man das Lock erhalten hat, aber nicht in dem Zeitraum davor. Man muss also sicherheitshalber nach dem Locking nochmal ans Dateiende gehen. Dazu verwendet man den Befehl `seek(FILEHANDLE, OFFSET, POS)`. Das dritte Argument kann `SEEK_SET` sein (ab Beginn der Datei), `SEEK_CUR` (ab der aktuellen Position) oder `SEEK_END` (ab dem Ende der Datei). Um ans Ende der Datei zu springen, verwendet man also den Befehl

```
1  seek(FILEHANDLE, 0, SEEK_END);
```

Statt `SEEK_SET`, `SEEK_CUR`, und `SEEK_END` kann man auch die Zahlen 0, 1, und 2 verwenden.

Der Code zum Anhängen an eine Datei sieht damit wie folgt aus:

```
1  use Fcntl qw(:DEFAULT :flock);
2
3  my $filename = "file.txt";
4  open OUT, ">>".$filename or die "can't open $filename: $!";
5  flock(OUT, LOCK_EX) or die "can't get a lock on $filename: $!";
6  seek(OUT, 0, SEEK_END);
7
8  print OUT "new text\n";
9
10 close(OUT) or die "$!"; # lock wird automatisch aufgehoben
```

Das Problem, dass ein anderer Prozess eventuell genau zwischen dem Öffnen und dem Sichern des Locks auf eine Datei noch dazwischenfunkt, ist ein typischer Fall einer so genannten "Race Condition". Das Abfangen möglicher Probleme durch Race Conditions ist eines der Hauptthemen bei der Beachtung von Sicherheitsaspekten bei der PERL-Programmierung.

15.5 (Über)Schreiben einer Datei

Beim Schreiben bzw. Überschreiben einer Datei (also das, was man normalerweise beim Öffnen eines Filehandles mit > bewirkt), fangen die Probleme schon beim Öffnen der Datei an: Wenn man die Datei mit `open FH, ">$filename"` öffnet, und sie existiert bereits, dann wird die Datei gelöscht, auch wenn ein anderer Prozess gerade liest oder schreibt, was Chaos verursachen würde. Andererseits kann man sich nicht vor oder beim Öffnen schon ein Lock auf die Datei sichern, sondern erst danach.

Die Lösung lautet in diesem Fall `sysopen(FILEHANDLE, DATEINAME, FLAGS)`. Mit diesem Befehl ist es möglich, eine Datei zum Schreiben zu öffnen, bzw. sie neu anzulegen, falls sie noch nicht existiert – der Unterschied zu `open FH, ">$filename"` ist aber, dass sie nicht beim Öffnen schon trunkiert wird, falls sie existiert. Als Flags gibt man in diesem Fall `O_WRONLY` und `O_CREAT` mit (*write only*, also nur schreibender Zugriff, und *create*, d.h. die Datei soll angelegt werden, wenn sie noch nicht existiert). Beide Flags werden mit binärem Oder (|) verknüpft. Das Programm sieht dann wie folgt aus:

```
1  use Fcntl qw(:DEFAULT :flock);
2
3  my $filename = "file.txt";
4  sysopen (OUT, $filename, O_WRONLY | O_CREAT) or die "can't open
       $filename: $!";
5  flock(OUT, LOCK_EX) or die "can't get a lock on $filename: $!";
6  truncate(OUT, 0);
7  # man trunkiert die Datei ab Byte 0 fuer den Fall, dass sie schon
       existiert
8  # aber erst, nachdem man den Exclusive Lock hat,
9  # das ist der entscheidende Unterschied zu open
10
11 print OUT "Inhalt\n";
12
13 close(OUT) or die "$!"; # Lock wird automatisch aufgehoben
```

15.6 Aktualisieren einer Datei

Das Aktualisieren einer Datei bedeutet, dass die Dateiinhalte erst ausgelesen und dann mit neuen Inhalten überschrieben werden. Auch hier bietet sich `sysopen` an. Man sichert sich für die Dauer des gesamten Vorgangs (lesen + schreiben) ein exklusives Lock. Nehmen wir als Beispiel einen Zähler, der von verschiedenen Prozessen hoch- oder runtergezählt werden soll, und der in einer Datei abgelegt ist (die nur eine einzige Zeile mit der Zahl enthält). Als Flags für `sysopen` wählen wir diesmal `O_RDWR` (Lese- und Schreibzugriff) und `O_CREAT` (die Datei soll angelegt werden, falls sie noch nicht existiert):

```
1  use Fcntl qw(:DEFAULT :flock);
2
3  my $filename = "counter.txt";
4  my $counter = 0;
5
6  sysopen (COUNTER, $filename, O_RDWR | O_CREAT) or die "can't open
       $filename: $!";
7  flock(COUNTER, LOCK_EX) or die "can't get a lock on $filename: $!";
8  $counter = <COUNTER> || ""; # Auslesen der ersten Zeile der Datei;
       wenn die Datei noch nicht existiert hat, ist der Rueckgabewert "
       undef"
9  chomp $counter;
10 seek(COUNTER,0,0) or die "Can't rewind counterfile: $!"; #
       Zurueckgehen zum Dateianfang
11 print COUNTER $counter+1, "\n";
12 close COUNTER or die "Can't close counterfile: $!";
```

In diesem speziellen Beispiel können wir sicher sein, dass der neue Dateiinhalt immer mindestens so lang oder länger ist als der alte Dateiinhalt. In Fällen, wo das nicht gewährleistet ist, ist folgende Zeile nach dem Schreiben sinnvoll:

```
1  truncate(COUNTER, tell(COUNTER)) or die "Can't truncate counterfile
       : $!";
```

Das bewirkt, dass die Datei an der aktuellen Position (ermittelt mit `tell(COUNTER)`) trunkiert wird, damit nicht eventuell noch ein Rest vom vorigen Inhalt hinter dem gewünschten Ende des neuen Eintrags steht.

15.7 File Locking und Forking

Werden vor einem `fork`-Befehl Filehandles geöffnet, stehen sie danach sowohl dem Parent- als auch dem Child-Prozess zur Verfügung. Ob allerdings eventuell existierende Locks auf dieses Filehandle nach dem Forken noch berücksichtigt werden, hängt vom benutzten Betriebssystem und der konkreten PERL-Implementierung ab – verlassen sollte man sich jedenfalls nicht darauf, dass Locks beim Forken korrekt dupliziert werden.

15.8 Übungsaufgaben

Aufgabe 15.1 Erweitere das parallel arbeitende Crawler-Programm aus Aufgabe 14.2 so, dass alle Prozesse die heruntergeladenen Seiten in dieselbe Datei schreiben (jeweils eine Trennzeile, eine Zeile mit der URL, dann eine Leerzeile und dann der Webseiteninhalt). Dabei soll File Locking angewendet werden, um Konflikte zwischen den verschiedenen schreibenden Prozessen zu verhindern. Bitte vor dem Abspeichern prüfen, ob auch tatsächlich erfolgreich etwas heruntergeladen wurde.

Aufgabe 15.2 Erweitere das Crawler-Programm aus Aufgabe 15.1 so, dass eine Zählerdatei eingerichtet wird. Am Anfang soll der Inhalt auf "0\n"gesetzt werden. Dann soll jeder Prozess, nachdem er erfolgreich eine Seite abgespeichert hat, den Zähler in der Datei hochzählen. Und am Schluss des Programms soll auf Basis der Zählerdatei ausgegeben werden, wieviele Webseiten insgesamt heruntergeladen worden sind.

16 WWW

16.1 Automatische Abfrage von Suchmaschinen mit Google::Search

Um aus einem PERL-Programm heraus eine Websuche mit Google durchzuführen, kann man das Modul `Google::Search` benutzen.

Installation

`Google::Search` kann einfach mit der CPAN-Shell installiert werden.

Benutzung

Hier ist ein einfaches Beispielprogramm, welches die Anfrage „CIS München" an das Google-API schickt und die zurückgegebenen Links ausgibt:

```
1 use Google::Search;
2
3 my $search = Google::Search->Web( query => "CIS Munchen" );
4 while ( my $result = $search->next ) {
5   print $result->rank, " ", $result->uri, "\n";
6 }
```

In Zeile 3 wird die Suche gestartet. Der Befehl ist eine Abkürzung für den Befehl:

```
1 my $search = Google::Search->new( query => "CIS Munchen", service
    => "web" );
```

Das Ergebnis ist dementsprechend ein Objekt der Klasse Google::Search. Dessen Inhalt wird mit der Methode `next` Treffer für Treffer ausgelesen. Jeder Aufruf von `next` liefert ein Objekt der Klasse Google::Search::Result. Die Methoden `rank` und `uri` liefern für jeden Treffer seinen Rank in der Trefferliste und die URL.

Auf dieselbe Weise kann man auch die Google-Services Video, Blog, News, Book, Image, Patent nutzen. Um nach Bildern zu suchen, kann man beispielsweise die Methode „Book" der Klasse Google::Search verwenden (bzw. beim Befehl „new" den Service „book" wählen).

Leider liefert das Google-API nur die ersten 64 Treffer. Außerdem hat Google das AJAX Search API, das Google::Search verwendet, als *deprecated* erklärt. Es wird daher möglicherweise irgendwann nicht mehr unterstützt. Für die Bing-Suchmaschine gibt es mit Bing::Search ein mit Google::Search vergleichbares Modul. Man muss allerdings eine ID beantragen, bevor man es nutzen kann.

16.2 wget

Alle Kursteilnehmer sollten inzwischen selbständig ein komplexeres Crawler-Programm schreiben können, das an die jeweiligen Bedürfnisse angepasst ist (Abarbeiten einer URL-Liste oder wirkliches

Crawlen mit Link-Weiterverfolgung, Einsatz paralleler Prozesse, Einstellen eines Timeouts, ...).

In vielen Fällen reicht jedoch auch ein Unix-Tool namens `wget`. Natürlich kann man es nicht beliebig anpassen, aber die Optionen eröffnen doch recht viele Möglichkeiten. Es kann also je nach Ziel durchaus sinnvoll sein, schnell auf das Unix-Tool zurückzugreifen, statt selbst einen Crawler in PERL zu schreiben. Deshalb soll diese Alternative hier vorgestellt werden.

Aufruf ohne Optionen

Ruft man das Programm ohne besondere Optionen auf, muss man als einziges Argument eine URL mitgeben. Ein Aufruf sieht dann z.B. so aus:

```
wget http://www.cis.uni-muenchen.de/~schmid/
```

Die Ausgabe auf dem Bildschirm ist wie folgt:

```
--2014-04-11 15:56:59--  http://www.cis.uni-muenchen.de/~schmid/
Auflösen des Hostnamen »www.cis.uni-muenchen.de (www.cis.uni-muenchen.de) ...
Verbindungsaufbau zu www.cis.uni-muenchen.de (www.cis.uni-muenchen.de) ...
HTTP-Anforderung gesendet, warte auf Antwort... 200 OK
Länge: 14803 (14K) [text/html]
In »»index.html«« speichern.

100%[=============================================>] 14.803      --.-K/s   in 0s

2014-04-11 15:56:59 (161 MB/s) - »»index.html«« gespeichert [14803/14803]
```

Wie man sehen kann, hat wget die Datei `index.html` gesucht und unter diesem Namen auch abgespeichert. Da keine weiteren Optionen (insbesondere nicht `-r` für "rekursiv") mitgegeben wurden, ist das Programm damit auch schon wieder beendet.

Ruft man das Programm nicht-rekursiv auf, versucht wget auf jeden Fall, die Seite zu holen, und zwar unabhängig davon, was `robots.txt` auf dem betreffenden Server sagt.

Rekursives Herunterladen

Damit sollte dann auch schon klar sein, wie man rekursiv Seiten herunterlädt: indem man zusätzlich `-r` als Option mitgibt. Für die Startseite gilt immer dasselbe wie für eine einzelne Seite: `wget` versucht sie herunterzuladen, egal was `robots.txt` sagt. Das gilt dann aber nicht mehr für weitere Seiten: hier gibt es keinen Weg für den Benutzer, `robots.txt` zu ignorieren, das wird von `wget` automatisch beachtet.

Während im ersten Beispiel einfach die Datei ohne den gesamten URL-Pfad abgespeichert wurde, wird (wenn man nichts Gegenteiliges angibt) beim rekursiven Herunterladen die Verzeichnisstruktur des Servers lokal nachgebildet. Wenn man also z.B. `wget -r www.cis.uni-muenchen.de` aufruft, so legt wget lokal ein Verzeichnis `www.cis.uni-muenchen.de` an, ggfs. weitere Unterverzeichnisse und speichert die geholten Seiten entsprechend ihrem Ort auf dem Server in diese lokale Verzeichnisstruktur.

Mit `-l Tiefe` kann man eine maximale Tiefe (innerhalb der Verzeichnisstruktur auf dem Server) angeben, bis zu der Dateien heruntergeladen werden. Default ist 5.

Laden aus einer Liste von URLs

Auch das ist möglich mit der Option `-i dateiname`. Die Datei "dateiname" enthält eine Liste von URLs.

Statt einer Datei kann man als Argument der Option auch eine URL angeben. Dann lädt wget diese Datei herunter, extrahiert alle Links und lädt auch diese herunter. Eine rekursive Extraktion findet nicht statt.

Bildschirmmeldungen

Per default wird `wget` im Verbose-Mode gestartet. Wem die Meldungen zu viel sind, hat folgende Möglichkeiten:

-o logfile: leitet die Meldungen in die genannte Datei um

-a logfile: hängt sie an die genannte Datei an

-q quiet, keine weiteren Bildschirmmeldungen

-nv non-verbose, nicht mehr so viele Meldungen, nur noch wichtigere

Das Abspeichern

Alternativ zum Default-Verhalten (s. oben) kann man auch alle heruntergeladenen Dateien in eine einzige Datei schreiben: `-O dateiname`. Speichert man hingegen in eine hierarchische Dateistruktur und unterbricht an irgendeinem Punkt, ist die Option `-nc` ("no clobber") sehr sinnvoll: zuerst werden die vorhandenen Seiten geparst und dann die noch fehlenden heruntergeladen (statt alles nochmal herunterzuladen und zu überschreiben).

Eine weitere Alternative ist `-nd` ("no directories"): alle Dateien werden in das aktuelle Verzeichnis geschrieben, ohne eine hierarchische Verzeichnisstruktur anzulegen. Werden mehrere Dateien desselben Namens heruntergeladen, werden schon vorhandene Dateien nicht überschrieben. Stattdessen wird an die Namen der neu heruntergeladenen Dateien eine Indexnummer angehängt.

Versuche und Wartezeiten

Mit `-t Anzahl` kann man steuern, wie oft `wget` versuchen soll, eine Seite herunterzuladen, ehe es aufgibt. `0` oder "inf" bedeuten unendlich viele Versuche.

Wie bereits erwähnt wurde, achtet wget beim Herunterladen von mehr als einer Seite automatisch auf `robots.txt`. Die andere wichtige Netiquette-Regel, nämlich wie lange `wget` warten soll, ehe es die nächste Seite holt, kann man mit `-w Sekunden` selbst einstellen. Mit den entsprechenden Suffixen (`m` für Minuten, `h` für Stunden und `d` für Tage) kann man auch größere Zeiteinheiten einstellen.

Ein- und Ausschlusskriterien

Beim rekursiven Herunterladen macht es Sinn, mittels verschiedener Kriterien festzulegen, welche Dateien bzw. welche nicht heruntergeladen werden sollen.

-A Suffixliste: eine Komma-getrennte Aufzählung von Dateiendungen, die heruntergeladen werden sollen ("accept");

-R Suffixliste: Dateiendungen, die nicht heruntergeladen werden sollen ("reject");

-H auch Webseiten von beliebigen anderen Hosts werden heruntergeladen, nicht nur von dem Host, auf dem die Startseite liegt (ohne zusätzliche andere Optionen wenig sinnvoll – in vielen Fällen würde das bedeuten, praktisch das ganze Web herunterladen zu wollen). Defaultmäßig ist das Herunterladen auf die Domain der Startseite beschränkt.

-D Domainliste: nur Dateien von dieser Domain sollen heruntergeladen werden. Sinnvoll in Kombination mit `-H`.

Weitere Optionen

Dies war nur ein kurzer Überblick über die wichtigsten Optionen. Eine ausführlichere Beschreibung befindet sich, wie für einige andere GNU-Tools auch, unter `http://www.gnu.org/manual/manual.html`.

16.3 HTML::Parser

Nachdem wir das Thema Crawlen recht ausführlich behandelt haben, soll es jetzt darum gehen, wie man die heruntergeladenen Webseiten weiterbearbeiten kann. Ihr habt schon `HTML::LinkExtor` kennengelernt, mit dem man die Links aus einem HTML-Dokument extrahieren kann. `HTML::LinkExtor` ist eine spezialisierte Subklasse von `HTML::Parser`. Wenn man aber auch andere Informationen als nur die Links aus einem HTML-Dokument extrahieren will, bietet es sich an, `HTML::Parser` selbst zu verwenden.

Es gibt zwei verschiedene Arten, mit `HTML::Parser` zu arbeiten. Eine ist als „Version 2", die andere als „Version 3" bekannt. Wir werden hier erstmal die Nutzung im Stil von „Version 2" behandeln.

Diese Art der Nutzung von `HTML::Parser` besteht darin, dass man selbst eine Subklasse schreibt. Alles, was man dort nicht an Methoden selber definiert, wird automatisch von `HTML::Parser` übernommen. Auf diese Art kann man sehr einfach die Klasse an die eigenen Bedürfnisse anpassen. (Dieses Prinzip ist allgemeingültig für die objektorientierte Programmierung, nicht nur für `HTML::Parser`).

Als Beispiel nehmen wir an, dass wir aus HTML-Dokumenten alle Links extrahieren wollen, genauer gesagt, die URLs der Webseiten, auf die verwiesen wird, und dazu der Text, mit dem auf diese externe Webseite verwiesen wird. Also z.B. `http://www.cis.uni-muenchen.de` und „Centrum für Informations- und Sprachverarbeitung". (Wenn es nur um die reinen Links ginge, wäre das LinkExtor-Modul praktischer, aber damit würden wir nicht an die zugehörigen Texte kommen).

Das Modul, dass die Subklasse enthält, würde so aussehen:

```
1  #!/usr/bin/perl
2
3  use strict;
4  use warnings;
5
6  package AnchorTextParser;
7  # so nennen wir das Modul und damit auch die Subklasse
8
9  use base 'HTML::Parser';
10 # hiermit sagen wir PERL, dass es sich um eine Subklasse von
11 # HTML::Parser handelt, d.h. dass alle nicht explizit in diesem
       Modul
12 # deklarierten Methoden von HTML::Parser uebernommen werden sollen
13
14 # hier wuerde man Variablen deklarieren, die im gesamten Modul
       gueltig sein sollen
15
```

```
16 sub ...
17                              # verschiedene Methoden
18 sub ....
19
20 1; # Abschluss des Moduls
```

Im aufrufenden Programm wären die entscheidenden Zeilen zunächst folgende:

```
1  #!/usr/bin/perl
2
3  use strict;
4  use warnings;
5
6  use AnchorTextParser;
7  # Einbinden des Moduls mit der Subklasse
8
9  my $parser = AnchorTextParser->new();
10 # Erzeugen eines Objekts dieser Subklasse
```

Danach sollte man dann dem Parser (ebenfalls im aufrufenden Programm) sagen, bei welchen Tags er aktiv werden soll (denn andernfalls wird er bei allen aktiv!):

```
1  $parser->report_tags("a");
2  # aktiv werden soll er nur bei Links, alle anderen Tags sollen
       ignoriert werden.
```

Nach diesen Vorbereitungen kann man dann einzelne Dokumente parsen:

```
1  $parser->parse($response->content); # hier parsen wir eine Seite,
       die von LWP::RobotUA geholt wurde
```

Was genau der Parser machen soll, wenn er beim Parsen an das nächste Element kommt (z.B. ein öffnendes oder ein schließendes Tag, Text, einen Kommentar etc.), kann man in der Subklasse mit den entsprechend benannten Methoden definieren (`start` definiert, was passiert, wenn beim Abarbeiten der `parse`-Methode ein öffnendes Tag identifiziert wird, `end` was bei einem schließenden Tag passiert, entsprechend `text`, `declaration`, `comment`, `process`, `start_document`, `end_document` und `default`). In unserem Fall beschränken wir uns auf das Schreiben der Methoden `start`, `end` und `text` (alle müssen in unserer Subklasse definiert werden).

An diese 3 Methoden werden von der `parse`-Methode automatisch folgende Argumente übergeben (s. `perldoc HTML::Parser`):

```
An "start": self, tagname, attr, attrseq, text
An "end": self, tagname, text
An "text": self, text, is_cdata
```

Bei der `start`-Methode wissen wir den Tag-Namen ja schon, da wir durch die Anweisung `$parser->report_tags("a")` vorher schon festgelegt haben, dass nur "a"-Tags einen Aufruf von start bewirken sollen. Das einzige für uns interessante Argument, das von der parse-Methode an die `start`-Methode übergeben wird, ist `attr`. Dies ist eine Referenz auf ein Hash mit allen Attribut-Wert-Paaren, die das öffnende Tag enthält. Der Anfang unserer `start`-Methode in der Subklasse würde also so aussehen:

```
1  my (undef, undef, $attr) = @_; # nur die Hash-Referenz mit den
       Attribut-Wert-Paaren interessiert uns
```

Danach kann man dann absichern, dass der Link auch tatsächlich auf eine Webadresse verweist:

```
1  return unless $attr->{href};
2  return if $attr->{href} =~ /mailto/i;
```

Erst wenn dies abgesichert ist, würde man dann z.B. eine (in der gesamten Subklasse gültige) Variable auf 1 setzen, die anzeigt, dass man tatsächlich ein Tag der gewünschten Art gefunden hat:

```
1 $in_anchor = 1;
```

Zusätzlich sollte man sich dann noch die in `$attr->{href}` enthaltene Webadresse (also die Link-Adresse) speichern – in eine Variable, die im gesamten Modul gültig ist. Das war dann schon alles, was in der `start`-Methode passieren sollte.

In der `end`-Methode (die in unserem Fall jedesmal bei einem aufgerufen wird) sollte man lediglich `$in_anchor` wieder auf 0 setzen, das ist dort schon alles. Auf diese Weise kann man in jedem Moment über `$in_anchor` abprüfen, ob man sich beim Parsen gerade zwischen einem öffnenden und einem schließenden Anchor-Tag befindet oder nicht.

In der `text`-Methode kommen alle möglichen Text-Stellen aus der HTML-Seite an – nicht etwa nur die Texte innerhalb der ausgewählten Tags. Hier interessiert uns nur das zweite übergebene Argument, nämlich der Text selbst:

```
1 my (undef, $origtext) = @_;
```

Als nächstes würde man dann absichern, dass man tatsächlich einen Link-Text erhalten hat:

```
1 return unless $in_anchor;
2 # Abbruch, wenn es sich nicht um Text handelt, der zwischen einem
      oeffnenden und einem schliessenden Anchor-Tag befindet
```

Danach sollte man mit

```
1 my $link_text = decode_entities($origtext); # dazu muss man das
      Perlmodul HTML::Entities einbinden
```

die HTML-Entitäten auflösen und den erhaltenen Text in ein Hash abspeichern, so dass er der zugehörigen Linkadresse zugeordnet ist (die Linkadresse hatten wir ja in der `start`-Methode in einer Variablen gespeichert).

Zusätzlich sollte man dann noch eine eigene Methode definieren, die die gesammelte Information zurückgibt, also z.B. so etwas wie

```
1 sub get_anchor_texts {
2   return \%anchor_texts;
3   # in %anchor_texts sollten nach dem Parsen die Linkadressen und
        die jeweils zugehoerigen Linktexte stehen
4 }
```

Im aufrufenden Programm braucht man dann nach dem Parsen des Dokuments nur noch diese Methode der Subklasse aufzurufen und die erhaltene Information auszugeben.

16.4 Übungsaufgaben

Aufgabe 16.1 Installiere `Google::Search` und lasse das Beispielprogramm laufen. Abzugeben: die Trefferliste zu einem selbstgewähltem Suchbegriff.

Aufgabe 16.2 Lade alle .jpg-Dateien von `www.cis.uni-muenchen.de` herunter, mit jeweils 3 Sekunden Wartezeit und ohne explizite Beschränkung der Verzeichnistiefe. Warum werden wohl so viele HTML-Seiten heruntergeladen?

Aufgabe 16.3 Schreibe ein Programm, dem als Argument eine Webseite mitgegeben wird. Diese Webseite soll dann geparst werden, wobei alle Linktexte ihren Linkadressen zugeordnet werden. Eine Übersicht über die Linkadressen und ihre Texte soll ausgegeben werden.

17

Pattern Matching

In diesem Kapitel wird das Thema Pattern Matching noch einmal in einigen Punkten vertiefend behandelt.

17.1 Funktionsprinzipien

Gefräßigkeit (Greediness)

Nehmen wir als Ausgangspunkt eine XML-Zeile

```
<tok><sur>esse</sur><lem cat="v" mor="1sp">essen</lem></tok>
```

Diese Zeile ist dem Format eines Tagger-Outputs entnommen, d.h. in ihr werden laufende Wörter eines Textes mit Lemmaform, Wortart und morphologischer Information annotiert. Umschließendes Tag ist `<tok> ... </tok>` (Token). Darin sind dann wiederum die beiden Tags `<sur>...</sur>` und `<lem..> ... </lem>` enthalten, die jeweils die Form des Wortes so enthalten, wie es im Text vorkommt (*sur*) bzw. die Lemmaform (*lem*). Bei der Lemmaform gibt es noch die Attribute cat für die Wortart (*category*) und für die morphologische Information (hier 1. Person Singular im Präsens).

Man könnte nun versuchen, mit dem folgenden Befehl die Wortart (Attribut `cat`) herauszufiltern:

```
1  $line =~ /cat="(.*)"/;
2  $wortart = $1;
```

Das Problem ist, dass man dann in `$wortart` nicht etwa `intj` stehen hat, sondern `intj" mor="1sp`. Das passiert, weil PERL beim Operator $*$ immer versucht, die längstmögliche Zeichenkette zu erwischen, so dass der Gesamtausdruck noch erfüllt wird. `cat="` kommt nur einmal vor, aber vom hinter den runden Klammern geforderten Anführungszeichen gibt es mehrere. Da durch den Punkt jedes beliebige Zeichen repräsentiert wird, nimmt PERL das letzte Vorkommen in der Zeile, da das dann den längsten Match ergibt.

Für dieses Problem gibt es zwei verschiedene Lösungen:

1. Man ersetzt den Punkt durch `[^\"]`. Das bedeutet, dass jedes Zeichen gematcht werden kann, außer einem Anführungszeichen. Der komplette Ausdruck würde dann so aussehen:

   ```
   1  $line =~ /cat="([^\"]*)"/;
   ```

   Damit stellt man sicher, dass auf keinen Fall die schließenden Anführungszeichen oder sogar noch weiteres darüber hinaus in $1 landen kann. Und Anführungszeichen dürfen bei XML in einem Attribut-Wert selbst nie unkodiert vorkommen.

2. Man kann auch die Gefräßigkeit der Operatoren beschränken. Während $*$, $+$ und ? versuchen, so viele Zeichen wie möglich zu matchen, versuchen $*?$, $+?$ und ?? so wenige Zeichen wie möglich zu matchen (immer unter der Voraussetzung, dass der Gesamtausdruck erfüllt werden kann). Diese zweite Lösung würde dann so aussehen:

```
1 $line =~ /cat="(.*?)"/;
```

Null-Vorkommen

Eine andere potenzielle Falle ist die Tatsache, dass der ∗-Operator auch bei Null-Vorkommen matcht – d.h. manchmal matcht er nicht unerwartet viel, sondern unerwartet wenig:

```
1 $string = "hallo";
2 print "match erfolgreich!\n" if $string =~ /b*/;
```

Tatsache ist, dass die Erfolgsmeldung ausgegeben wird, obwohl "hallo" offensichtlich keinerlei "b" enthält. Da aber der ∗-Operator besagt „Null oder mehr Vorkommen" ist der reguläre Ausdruck auch bei "hallo" erfüllt.

Eagerness

Als Eagerness bezeichnet man den Umstand, dass PERL beim Pattern-Matching versucht, so schnell wie möglich einen Match zum Erfolg zu bringen. Im obigen Beispiel mit `/b*/` und "hallo" bedeutet das, dass schon beim Abprüfen der ersten Position in "hallo" die Match-Bedingung als erfüllt betrachtet wird. Ein anderes Beispiel macht es noch deutlicher:

```
1 $string = "zimmer";
2 $string =~ s/m*/x/;
3 print "$string\n";
```

Ein etwas überraschendes Ergebnis: "xzimmer". – Warum? PERL hat die erste Stelle genommen, an der die Match-Bedingung erfüllt war, und das war die erste Position im String. Dort wurde dann das "x" eingefügt.

Verwendet man bei der Ersetzung die Option g für "globales Ersetzen", d.h. Ersetzen an allen vorkommenden Stellen (also `$string =~ s/m*/x/g`), dann wird an JEDER Position ein "x" eingefügt, da an jeder Position die Bedingung erfüllt ist. Steht an der Position tatsächlich ein oder mehrere "m", dann werden sie natürlich gematcht, da kommt dann wieder die Gefräßigkeit durch (2 Buchstaben werden gematcht statt null). Das Ergebnis lautet:

`xzxixxexrx`

Denn die Bedingung ist erfüllt:

- beim Überprüfen des "z"
- beim Überprüfen des "i"
- beim Überprüfen der 2 "m" (die effektiv ersetzt werden)
- beim Überprüfen des "e"
- beim Überprüfen des "r"
- und nochmal ganz am Ende des Strings.

Greediness vs. Eagerness

In gewisser Weise widersprechen sich Greediness und Eagerness – einerseits versucht PERL immer den längstmöglichen Match zu finden, andererseits immer den erstmöglichen. Diese Kombination ist im Allgemeinen bekannt als "leftmost longest match". D.h. PERL geht von links nach rechts durch den String, und versucht an jeder Stelle, den regulären Ausdruck erfolgreich zu matchen. Sobald an einer Position der reguläre Ausdruck erfüllt werden kann, wird das Ergebnis zurückgegeben, weiter

rechts liegende Möglichkeiten werden nicht durchprobiert (die Eagerness führt dazu, dass immer der am weitesten links beginnende Treffer zurückgegeben wird, daher "leftmost"). Innerhalb dieser Versuche kommt dann die Greediness zum Zug: es wird immer versucht, so viele Zeichen wie möglich zu matchen, wenn es mehrere Möglichkeiten gibt (z.B. bei $*$ "null bis mehrere Vorkommen", sofern man nicht die Greediness explizit abgeschaltet hat durch Verwendung von $*?$). Daher das "longest" in "leftmost longest match".

Die Tatsache, dass die Greediness nur innerhalb der Regeln der Eagerness angewendet wird, kann dazu führen, dass nicht der insgesamt längstmögliche Match zurückgegeben wird – nämlich genau dann, wenn PERL schon vorher eine Lösung gefunden hat.

```
1 my $string = "abracadabra";
2 $string =~ /(a|cadabra)+/;
3 print "$1\n";
```

In $1 findet sich "a", weil es direkt am Anfang des Strings zu einer Lösung führt, obwohl später noch ein längerer Treffer hätte gefunden werden können. Das ist aber gar nicht mehr ausgetestet worden.

Backtracking

Das dritte und letzte Prinzip, das man verstehen muss, um Matching-Ergebnisse genau vorhersagen zu können, ist das Backtracking.

Beispiel:

```
1 my $string = "abracadabra";
2 $string =~ /(a|ab)ra/;
3 print "$&\n"; # $& enthaelt den Gesamtmatch
```

PERL startet beim ersten Buchstaben des Strings, "a", testet die erste im regulären Ausdruck vorgegebene Möglichkeit des Matches, nämlich das dortige "a" und kommt zu einem positiven Ergebnis (Alternativen im regulären Ausdruck werden immer von links nach rechts durchprobiert, nicht etwa ihrer Länge nach). Dann versucht es noch den Teil "ra" des regulären Ausdrucks zu erfüllen – und scheitert. An diesem Punkt geht PERL dann zur letzten Auswahlmöglichkeit zurück, stellt fest, dass auch "ab" matcht, versucht dann nochmal "ra" zu matchen – und kommt erfolgreich zum Gesamtergebnis "abra".

Da PERL es auch erlaubt, PERL-Code in reguläre Ausdrücke einzubinden, kann man sich sogar die Zwischenstände des Matches anzeigen lassen, um die Abarbeitung nachzuvollziehen:

```
1 my $string = "abracadabra";
2 $string =~ /(a|ab)(?{print "Vorlaeufiger Gesamtmatch in diesem
     Moment: $&\n";})ra/;
3 print "Gesamtmatch am Ende: $&\n";
```

Der PERL-Code wird dabei jedesmal ausgeführt, wenn PERL im Rahmen des Match-Prozesses an der Stelle „vorbeikommt", wo der PERL-Code steht. Der PERL-Code muss immer von `(?{` eingeleitet und von `})` abgeschlossen werden. Es muss allerdings darauf hingewiesen werden, dass diese Möglichkeit nur als „experimentell" implementiert ist.

Dieses Backtracking findet immer im Rahmen des bei „Greediness vs. Eagerness" beschriebenen Gesamtverfahrens statt, d.h. die Positionen werden im String von links nach rechts abgeprüft und es wird nur weitergegangen, wenn sich von dieser Startposition aus der Gesamtausdruck nicht irgendwie (ggfs. unter Anwendung von Backtracking) erfüllen lässt.

17.2 Nützliche Optionen

Die Option "i" wird als bekannt vorausgesetzt und deshalb hier nicht behandelt (ebenso wie „e" und „ee", weil sie nur bei Ersetzungen anwendbar sind und „cg" weil es nicht sehr häufig benötigt wird). Die Option „o" wird im Zusammenhang mit dem Thema der effizienten Programmierung behandelt.

Mehrfach im selben String matchen mit "g"

Manchmal tritt das Problem auf, dass man in derselben Zeile mehrfach denselben regulären Ausdruck abprüfen möchte. Nehmen wir an, in einer Zeile wie oben gäbe es eventuell mehrere `lem`-Tags, d.h. es werden mehrere mögliche Wortarten annotiert. Eine solche Zeile könnte z.B. so aussehen:

```
<tok><sur>Bayern</sur><lem cat="en" mor="">bayern</lem><lem cat="n" mor="nmM">bayer</lem></tok>
```

(d.h. einmal Bayern als Eigenname des Bundeslandes und einmal als Nominativ Plural des maskulinen Wortes der Bayer).

Wie matcht man jetzt nacheinander die beiden `<lem...>...</lem>` Informationen?

Natürlich kann man ein bis mehrere `<lem ...> ... </lem>` – Einträge mit so einem Ausdruck einfangen:

```
1   /(<lem.*?<\/lem>)+/; # auch hier ist das Fragezeichen wichtig!
```

Aber das ist nicht besonders praktisch, weil man danach immer noch analysieren muss, ob man denn jetzt ein oder mehrere `lem`-Informationen gematcht hat und den Treffer ggfs. nochmal splitten muss.

Die günstigere Lösung ist eine `while`-Schleife, wobei man beim Matchen die Option "g" nutzt:

```
1  while ($line =~ /(<lem.*?<\/lem>)/g) {
2      $lem = $1;
3      ....
4  }
```

Normalerweise wird nach einem Treffer die Position im String, bei der der nächste Matchversuch starten soll, wieder auf den Anfang des Strings gesetzt. Die Option "g" verhindert das, so dass bei mehrfacher Anwendung des Matchings die Suche jeweils unmittelbar hinter dem Treffer von der letzten Anwendung beginnt.

Diese nächste Startposition für einen Match kann man übrigens mit der pos-Funktion auch explizit abfragen, z.B. `pos($line)`im obigen Beispiel – oder man kann die Startposition auch explizit zuweisen, z.B. mit `pos($line)= 4`. Dann beginnt die Suche erst hinter dem 4. Zeichen im String. Innerhalb eines regulären Ausdrucks kann man mit `\G`' auf diese nächste Startposition referieren, so dass so etwas möglich wird:

```
1  my $line = "Mein Hut, der hat 3 Ecken";
2  $line =~ /\bhat /g;
3  print $1 if $line =~ /\G(\d+ \w+)/;
```

Hier wird überprüft, ob unmittelbar hinter dem ersten Wort "hat" und einem Leerzeichen eine Zahl gefolgt von einem Wort kommt. Würde der String lauten "Mein Hut, der hat ne Hutschnur und mein Tisch, der hat 3 Ecken", so würde im obigen Beispiel nichts gematcht. Lässt man hingegen das `\G` im regulären Ausdruck weg, dann würde bei beiden Beispielsätzen "3 Ecken" gematcht, also egal ob der Match direkt an "hat " anschließt oder erst irgendwo viel weiter hinten kommt.

Reguläre Ausdrücke übersichtlicher schreiben mit "x"

Die Option "x" bewirkt, dass Leerzeichen inkl. Zeilenumbrüche im regulären Ausdruck nicht beachtet werden. Damit kann man einen komplexen regulären Ausdruck über mehrere Zeilen verteilt schreiben und Leerzeichen zur Gliederung benutzen. Man kann sogar Kommentare ans Ende der Zeilen schreiben, so dass die regulären Ausdrücke sehr viel leichter nachvollziehbar werden.

Beispiel:

```
1 $line =~ /<lem      # start of the lemma tag
2           .*?>      # attributes are not of interest in this case
3           (.*?)     # getting the lemma form
4           <\/lem>   # end of the lemma tag
5           /x;
6
7 $lemma = $1;
```

In diesem Beispiel ist das noch relativ trivial, aber sobald man mit komplexeren regulären Ausdrücken arbeitet, ist es absolut empfehlenswert.

Matchen auf mehreren Zeilen: "m" und "s"

Manchmal kommt es vor, dass man in der String-Variablen, auf der man einen Match durchführt, mehrere Zeilen Text hat. Dies kann z.B. passieren, wenn man einen kompletten Dateiinhalt auf einmal in eine String-Variable einliest:

```
1 local($/);
2 undef($/);
3 open (IN, "text.txt") or die "$!";
4 my $text = <IN>;
```

Wenn man auf solch einem String dann mit "." matcht, ist immer an einem Zeilenende Schluss, weil der Punkt alles matcht außer einem Zeilenumbruch. Wenn man jedoch die Option "s" wählt, dann matcht der Punkt auch Newlines und

```
1 $text =~ /Perlstunde.*?Erfolgserlebnis/s;
```

matcht auch dann, wenn zwischen "Perlstunde" und "Erfolgserlebnis" noch ein paar Zeilenumbrüche liegen.

Es ist jedoch ein Irrtum, wenn man glaubt, man müsste immer die Option "s" aktivieren, wenn man auf einem mehrzeiligen String etwas matchen will. Das geht auch ohne die Option "s" völlig problemlos. Der wirklich einzige Unterschied, den die Option "s" bewirkt, ist, ob der Punkt auch Newlines matcht oder nicht.

Enthält eine String-Variable Zeilenumbrüche, dann matchen ^ und $ trotzdem nur am Beginn des Strings und am Ende. Das kann man mit der Option "m" ändern: dann matchen ^ und $ zusätzlich hinter bzw. vor jedem in dem String vorkommenden Zeilenumbruch.

17.3 Backreferences und Lookaround-Bedingungen

Backreferences

Wie bekannt sein dürfte, kann man mit `$1 .. $n` bei erfolgreichem Match die Matches von in runden Klammern eingeschlossenen Teilpattern abfragen. Dies funktioniert jedoch nicht innerhalb des regulären Ausdrucks selbst, wenn man z.B. kontrollieren will, ob irgendwo im String zweimal unmittelbar hintereinander dasselbe Wort steht:

```
1 $line =~ /\b(\w+) $1\b/; # falsch!
```

Das heißt aber nicht, dass das unmöglich wäre. Will man innerhalb des selben Strings Backreferences benutzen, muss man einfach statt `$1 .. $n` `\1 .. \n` abfragen, also:

```
1 $line =~ /\b(\w+) \1\b/; # richtig
```

Ein vielleicht häufigeres Problem besteht darin, dass man einen Haufen runder Klammern in einem regulären Ausdruck verwendet, wobei man bei einigen die Backreferences abfragen will, bei anderen aber einfach nur Alternativen oder ähnliches klammern will – und dann wird es sehr unübersichtlich, wenn man plötzlich noch die zu $7 gehörigen Klammern finden will. Bei dieser Komplexität empfiehlt sich auf jeden Fall schon mal die Option "x", aber abgesehen davon kann man auch verhindern, dass bestimmte Klammern überhaupt eine Backreference erzeugen: man muss unmittelbar hinter der öffnenden Klammer "?:" schreiben. Also z.B.:

```
1 $line =~ /(Mein \w+)(?:, der)? hat (\d+) Ecken/;
2 my $was = $1;
3 my $wieviele_ecken = $2;
```

Das zweite Klammerpaar, das ein optionales ", der" abprüft, wird für die Backreferences nicht mitgezählt.

Die Verwendung von ?: hat zwei wichtige Vorteile:

1. die Zählung der Backreferences verschiebt sich nicht, wenn man noch irgendwo zusätzlich ein Klammerpaar einfügt;
2. das Programm wird beschleunigt, da PERL nicht den Teilmatch zur Verfügung stellen muss.

Lookaround-Bedingungen

Lookahead

Das Fragezeichen unmittelbar hinter einer öffnenden Klammer leitet auch noch verschiedene andere Features ein. Eines davon ist das so genannte Lookahead – man will sicher sein, dass hinter dem Match etwas Bestimmtes kommt, aber dieses dahinterkommende etwas selbst will man aus dem Match heraushalten. Ein Beispiel wäre, wenn man aus einer Zeile Tagger-Output mit mehreren Lemma-Informationen alle Lemma-Informationen herausholen will mit Ausnahme der letzten.

Würde man alle Lemma-Informationen herausholen wollen, würde man einfach so etwas schreiben:

```
1 while ($line =~ /(<lem.*?<\/lem>)/g) {
2     $lemma_info = $1;
3     ....
4 }
```

Jetzt muss man aber sicherstellen, dass hinter dem schließenden Lemma-Tag wieder ein öffnendes Lemma-Tag kommt – nur dann soll der Match erfolgreich sein. Aber bei

```
1 while ($line =~ /(<lem.*?<\/lem>)<lem/g) {
2     ...
```

würde man das nächste `<lem` mitmatchen, und beim nächsten Schleifen-Durchlauf würde die Suche statt bei `<lem` genau dahinter starten und könnte diese Lemma-Information nie matchen.

Eine Möglichkeit zur Lösung wäre es, `pos($line)` jedesmal entsprechend zurückzusetzen. Die übersichtlichere Möglichkeit ist es aber, eine Lookahead-Bedingung einzubauen:

```
1 while($line =~ /(<lem.*?<\/lem>)(?=<lem)/g) {
2     ...
```

In diesem Fall gelingt der Match nur, wenn das hinter ?= stehende Muster erfolgreich gematcht werden kann – es selbst wird aber nicht mitgematcht und verschiebt auch die Startposition für die nächste Suche nicht.

Man kann das Ganze auch umgekehrt angehen und eine negative Bedingung formulieren: ich will die Lemma-Information genau dann, wenn dahinter etwas bestimmtes NICHT kommt. In vorliegenden Fall könnte ich statt eines folgenden <lem genauso gut fordern, dass kein </tok> folgen darf. Im Gegensatz zum oben verwendeten "positive lookahead" nennt man so etwas ein "negative lookahead". Eingeleitet wird es durch ein "?!" nach einer öffnenden Klammer. Die Lösung ließe sich also auch so formulieren:

```
1  while($line =~ /(<lem.*?<\/lem>)(?!<\/tok>)/g) {
2      ...
```

Lookbehind

Ähnliches kann man formulieren für etwas, was links von dem stehen soll, was man matchen will. Die Schreibweise ist ein "?<=" unmittelbar nach einer öffnenden Klammer. Nehmen wir an, ich will alle Lemma-Informationen matchen außer der ersten. Dann kann ich fordern, dass vor meiner gesuchten Lemma-Information ein schließendes Lemma-Tag stehen muss:

```
1  while($line =~ /(?<=<\/lem>)(<lem.*?<\/lem>)/g) {
2      ...
```

Und wieder gibt es auch eine negative Variante, die dann mit "?<!" eingeleitet wird:

```
1  while($line =~ /(?<!<\/sur>)(<lem.*?<\/lem>)/g) {
2      ...
```

Zusammenfassung

positiver Lookahead: (?=PATTERN)
negativer Lookahead: (?!PATTERN)
positiver Lookbehind: (?<=PATTERN)
negativer Lookbehind: (?<!PATTERN)

Sonstiges

Während die Pattern beim Lookahead beliebige reguläre Ausdrücke sein können, kann man beim Lookbehind nur Muster mit fester Länge schreiben. In ihnen darf also z.B. keiner der Operatoren "$*$ $+$?" vorkommen.

Eine interessante Eigenschaft aller Lookaround-Bedingungen ist übrigens, dass man Backreferences erhält, wenn man in den Lookaround-Mustern Klammern verwendet – obwohl ja die Lookaround-Muster eigentlich nicht zum Match gehören. Die Klammer, die für die Lookaround-Bedingung selbst verwendet wird, zählt dabei jedoch nicht mit, nur innerhalb dieser Bedingung vorkommende Klammern.

17.4 Berücksichtigung von lokalen Zeichensätzen

Dieses Thema ist für das Pattern Matching wichtig, aber genauso auch für Sortier-Befehle und anderes mehr.

`locale`-Konstanten

Das sind Systemkonstanten, die regeln, nach welcher Sprache sich Befehle und Formatierungen richten. Bsp: die in `LC_MONETARY` angegebene Sprache bestimmt das "monetary formatting", die in `LC_TIME` gesetzte Sprache "time and date formatting" (z.B. 24-h-System oder a.m. / p.m.). Die locale-Werte werden nur angewendet, wenn sie explizit vom Programm eingebunden werden!

LC_CTYPE

Legt fest, welche Codezeichen Buchstaben, Ziffern, Leerzeichen, alphanumerische Zeichen, druckbar usw. sind. Außerdem wird hier festgelegt, welche Klein- und Großbuchstaben zusammengehören. Diese Konstante wirkt sich direkt auf reguläre Ausdrücke aus! (z.B. ob Umlaute in `\w` mitberücksichtigt werden oder nicht)

LC_COLLATE

Legt fest, wie Wörter alphabetisch geordnet werden sollen. Wirkt sich auf `gt`, `lt`, `ge`, `le`, `cmp` aus, außerdem auf `sort`, falls keine explizite Unterroutine verwendet wird (die default-Sortierung geschieht mittels `cmp`, das von `LC_COLLATE` beeinflusst wird)

LC_ALL

Wird in `LC_ALL` eine Sprache festgelegt, so gilt sie als für alle locale-Konstanten festgelegt, d.h. nach dieser Sprache richten sich dann Datums- und Währungsformatierung, alphabetische Sortierung etc.

Die Berücksichtigung lokaler Zeichen erreichen

Die Lösung heißt im Prinzip einfach `use locale;` damit werden in `locale` gespeicherte Informationen (systemabhängig!) beim Kompilieren genutzt.

Das reicht bei den meisten Systemen schon, aber nicht ganz bei allen. Bei manchen wenigen Systemen ist möglicherweise `LC_CTYPE` nicht oder nicht korrekt gesetzt. Außerdem will man natürlich meist erreichen, dass ein Programm unabhängig von den Rechnereinstellungen immer gleich funktioniert. Dann muss man zusätzlich

```
1 use POSIX 'locale_h';
```

aufrufen und `LC_CTYPE` mit `setlocale()` setzen:

```
1 setlocale (LC_CTYPE, 'de_DE.ISO_8859-1')
2         or die "invalid locale";
```

Die verfügbaren locales kann man bei den meisten Systemen auf der Shell mit

```
locale -a
```

abfragen.

Das aktuell gültige `LC_CTYPE` in PERL abfragen kann man mit:

```
1 print setlocale(LC_CTYPE), "\n";
```

Den Befehl `use locale;` kann man auch nur in einem bestimmten Block des Programms verwenden:

```
1 @x = sort @y;    # ASCII -- Sortierung
2 {
3   use locale;
4   @x = sort @y;  # Sortierung entsprechend LC_COLLATE
5 }
6 @x = sort @y;    # ASCII -- Sortierung
```

Und hier noch ein kleines Beispiel, um die Unterschiede auf dem aktuellen Rechner zu testen:

```
1  #!/usr/bin/perl -w
2
3  use strict;
4  use POSIX 'locale_h'; # Damit ich fuer das Abfragen von LC_CTYPE
       die Funktion setlocale zur Verfuegung habe
5
6  print STDERR "Der aktuelle Wert von LC_CTYPE auf diesem Rechner ist
       ", setlocale(LC_CTYPE), "\n";
7
8  my $wort = <>;  # das Wort wird ueber STDIN uebergeben, z.B. ein
       Wort mit Umlaut
9  print STDERR "Ich ueberpruefe jetzt, ob das Wort '$wort' nur aus
       Wortzeichen besteht oder nicht.\n";
10 print STDERR "a) ohne explizite Einbindung der locales:\n";
11
12 if ($wort =~ /^\w+$/) {
13   print STDERR "Das Wort '$wort' enthaelt ausschliesslich
         Wortzeichen.\n";
14 } else {
15   print STDERR "Das Wort '$wort' enthaelt auch andere Zeichen als
         Wortzeichen.\n";
16 }
17
18 print STDERR "b) mit expliziter Einbindung der locales:\n";
19
20 {
21   use locale;
22   if ($wort =~ /^\w+$/) {
23     print STDERR "Das Wort '$wort' enthaelt ausschliesslich
           Wortzeichen.\n";
24   } else {
25     print STDERR "Das Wort '$wort' enthaelt auch andere Zeichen als
           Wortzeichen.\n";
26   }
27 }
```

Sonstiges zum Thema:

Weitere Informationen findet man mit `man perllocale`, `man iso_8859_1`, und `man utf8`.

17.5 Der Substitution-Operator

Der substitution-Operator (`s///`) als solcher wird als bekannt vorausgesetzt, ebenso die Optionen, die auch auf den Match-Operator (`m//`) anwendbar sind.

Die Option "e"

Normalerweise werden im zweiten Teil einer Substitution bereits Variablen ausgewertet, d.h. man kann so etwas schreiben wie

```
1  my $string = "This house is green";
2  my %wortart = ("this" => "det", "house" => "n", "is" => "v", "green
       " => "adj");
3  $string =~ s/\b(\w+)\b/$1\/$wortart{lc($1)}/g;
```

um in einem String alle Wörter um ihre Wortart zu ergänzen.

Manchmal jedoch möchte man etwas Komplizierteres ausdrücken. Dann kann man auch PERL-Code in den zweiten Teil der Substitution schreiben und die Option "e" verwenden. Auf diese Weise wird der PERL-Code ausgewertet ("e" = "evaluate"), und der Rückgabewert des PERL-Codes wird als Ersetzung genommen.

```
1 $string =~ s/\b(\w+)\b/{if ($wortart{lc($1)}) { "$1\/$wortart{lc($1
    )}"} else { "$1\/nicht im Lexikon!"}}/ge;
```

Wenn es noch komplexer wird, kann man auch einfach eine Subroutine aufrufen, die natürlich beliebig lang und komplex sein kann, und deren Rückgabewert dann die Ersetzung enthält:

```
1 $string =~ s/\b(\w+)\b/bestimme_wortart($1)/ge;
```

Es ist auch möglich, noch ein weiteres "e" als Option anzufügen. Dann wird die Rückgabe der ersten Auswertung wiederum als PERL-Code ausgewertet, und erst dieses Ergebnis als Ersetzung verwendet. Das lässt sich beliebig verschachteln – aber besonders sinnvolle Beispiele fallen mir dazu nicht ein.

s/// benutzen, um bestimmte Vorkommen im String zu zählen

Da der s///-Operator die Anzahl der durchgeführten Ersetzungen zurückgibt, kann man ihn auch zum Zählen von Vorkommen von irgendetwas im String verwenden.

```
1 $string = "abracadabra";
2 $anzahl = ($string =~ s/(abra)/$1/g);
3 print "$anzahl\n";
```

Das Programm gibt zurück, wie oft "abra" im String "abracadabra" vorkommt.

Mit dem Match-Operator funktioniert das nicht – der Match-Operator gibt in skalarem Kontext immer 1 oder 0 zurück (je nachdem ob der Match erfüllt werden kann oder nicht), und in Listenkontext die Werte von $1 bis $n.

17.6 Übungsaufgaben

Aufgabe 17.1 Eine einfache (und nicht allzu sichere) Art, einen Text zu verschlüsseln, besteht darin, jeden Buchstaben durch den im Alphabet folgenden Buchstaben zu ersetzen ("hallo" -> "ibmmp"). Und natürlich muss es nicht der direkt folgende sein, sondern es kann auch jeweils der xte Buchstabe dahinter sein. Geht die Berechnung dieses neuen Buchstabens über das Ende des Alphabets hinaus, fängt man vorne wieder an (z.B. Abstand = 3, "zyklop" -> "cbnors"). Realisiere diese Verschlüsselung mittels eines s///-Befehls, der die Option "e" verwendet. Der Buchstabenabstand soll vom Benutzer eingegeben werden. Umlaute brauchen nicht berücksichtigt zu werden.

Hinweise:

- Die PERL-Funktion `ord()` liefert den ASCII-Wert eines Buchstabens, `chr()` liefert den Buchstaben zu einem vorgegebenem ASCII-Wert.
- Der `substitute`-Befehl sollte nacheinander alle Buchstaben des eingegebenen Wortes durchgehen und ihn jeweils durch einen neuen Buchstaben ersetzen, am besten unter Einsatz einer Subroutine.
- Wird der ASCII-Wert des neuen Buchstabens größer als der ASCII-Wert von "z", dann sollte vom ASCII-Wert des neuen Buchstabens 26 abgezogen werden.

18

Beschleunigung von Perlprogrammen

18.1 Wie schnell ist mein Programm eigentlich?

Was man als Benutzer direkt beobachten kann, ist die Zeit, die das Programm braucht, bis es fertig ist. Diese Zeit ist aber nur bedingt aussagekräftig – denn sie hängt entscheidend davon ab, welchen Anteil an der Rechenleistung das Programm bekommen hat und welcher Anteil für andere Prozesse verwendet wurde.

Eine genaue Übersicht über diese Zahlen erhält man mittels des Unix-Kommandos `time`: es wird einfach vor das auszuführende Programm gesetzt und liefert nach Ende des Programms die Übersicht. Für ein PERL-Programm sieht der Aufruf z.B. so aus:

```
time perl zeitmessung.perl
```

und der Output:

```
perl zeitmessung.perl  12,27s user 0,00s system 99% cpu 12,349 total
```

Von rechts her gelesen bedeutet das: insgesamt ist das Programm 12,349 Sekunden gelaufen, wobei es 99 % der CPU-Leistung bekommen hat. Die Zeit, die das Programm verbraucht hätte, wenn es 100 % der CPU-Zeit gehabt hätte, ergibt sich aus den addierten Werten für `user` und `system`: 12,27 Sekunden. `user` ist dabei die Zeit, die vom Programm als solchem verbraucht wurde, mit `system` wird diejenige Zeit bezeichnet, die benötigt wird, wenn vom Programm her Routinen des Betriebssystems aufgerufen werden.
Diese Summe von User- und Systemzeit ist dann ein wesentlich objektiverer Maßstab dafür, wie schnell oder langsam das Programm wirklich ist.

18.2 Effizienzanalyse mit Profiler-Programmen

Analyse nach Subroutinen

Wichtigste Voraussetzung zu einer Beschleunigung von Perlprogrammen ist i.d.R. eine Analyse, wo im Programm wieviel Zeit verbraten wird. Diese Information liefern so genannte Profiler-Programme wie z.B. `Devel::DProf`. Man bindet es jedoch nicht über `use` ein, sondern ruft das zu kontrollierende Programm üblicherweise mit der Option `-d:DProf` auf, also z.B. `perl -d:DProf my_program.perl`. Das Programm läuft dann genauso durch, wie es das sonst auch tun würde, aber es wird eine Datei namens tmon.out angelegt, in der sich die für uns interessanten Informationen befinden.

Direkt lesbar ist diese Information nicht, aber das Programm dprofpp wandelt die Information in eine für Menschen lesbare Form um, der Aufruf lautet `dprofpp tmon.out`.

Hier ein (extrem sinnvolles!) Perlprogramm als Beispiel:

```
1  #!/usr/bin/perl
2
3  use strict;
4  use warnings;
5
6  #main
7  {
8      my @values = set_variables();
9      output(@values);
10 }
11
12 sub set_variables {
13
14     my $a = 7;
15     my $b = 5;
16     my $c = 13;
17     my $d = 0;
18
19     foreach my $x (0..1000000) {
20         my $y = sqrt($x);
21     }
22
23     foreach my $x (1..2) {
24         $d = set_d($c);
25     }
26
27     return ($a, $b, $c, $d);
28 }
29
30 sub set_d {
31     my $c = shift;
32
33     foreach my $x (0..1000000) {
34         my $y = sqrt($x);
35     }
36
37     return $c * $c;
38 }
39
40 sub output {
41
42     my @values = @_;
43
44     foreach my $value (@values) {
45         foreach my $x (0..1000000) {
46             my $y = sqrt($x);
47         }
48         print "$value\n\n";
49     }
50 }
```

Die Subroutine `set_variables()` wird einmal aufgerufen und ruft selbst zweimal die Subroutine `set_d()` auf. In beiden wird jeweils 1 Mio. mal eine Wurzel berechnet. `output()` wird nur einmal aufgerufen und berechnet nochmal pro Variable je 1 Mio. Wurzeln.

Das Ergebnis von `perl -d:DProf program.perl`, gefolgt von `dprofpp tmon.out` ist folgender:

```
Total Elapsed Time = 12.23982 Seconds
  User+System Time = 12.22982 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 56.9   6.960  6.960      1   6.9600 6.9600  main::output
 28.3   3.470  3.470      2   1.7350 1.7350  main::set_d
 14.2   1.740  5.210      1   1.7400 5.2100  main::set_variables
 0.25   0.030  0.030      2   0.0150 0.0150  main::BEGIN
 0.00   0.000 -0.000      1   0.0000      --  strict::import
 0.00   0.000 -0.000      1   0.0000      --  strict::bits
 0.00   0.000 -0.000      1   0.0000      --  warnings::BEGIN
 0.00   0.000 -0.000      1   0.0000      --  Exporter::import
 0.00   0.000 -0.000      1   0.0000      --  warnings::import
```

"Total Elapsed Time" entspricht dem Wert "total" beim Kommando "time", "User+System Time" sollte damit auch klar sein. Danach wird es dann erst richtig interessant mit den einzelnen Werten pro Subroutine (deren Name jeweils ganz rechts in der Zeile aufgeführt ist). Die Spalte ganz links ("% Time", die Überschriften sind hier leider etwas verrutscht) gibt an, welcher Anteil der verbrauchten Zeit für diese Subroutine insgesamt verwendet wurde (d.h. für alle Durchläufe der Subroutine zusammengenommen). "ExclSec" gibt an, wie viel Zeit dies absolut gesehen war. Bei diesen beiden ersten Werten werden jeweils die Subroutinen nicht mitgerechnet, die wiederum innerhalb der rechts genannten Subroutine aufgerufen werden – so wird z.B. für `set_variables()` diejenige Zeit nicht berücksichtigt, die in `set_d()` verbraucht wird.

Beim dritten Wert, "CumulS", hingegen wird auch die Zeit von aufgerufenen Subroutinen mitgerechnet, so dass sich hier der Wert für `set_variables()` deutlich erhöht. Es folgt die Anzahl, wie oft die Subroutine insgesamt aufgerufen wurde ("#Calls") sowie die verbrauchte Zeit pro einzelnem Aufruf ("sec/call"). "Csec/call" ("Cumulated seconds per call") gibt diesen Wert wieder unter Einbeziehung der aufgerufenen Subroutinen an.

Es bleibt festzuhalten, dass meistens der erste und der letzte Eintrag der ersten Zeile des Zahlenteils am interessantesten sind: sie geben an, in welcher Subroutine die meiste Zeit verbraucht wurde und wo somit vermutlich das größte Beschleunigungspotential liegt.

Es gibt übrigens noch eine Reihe von Komandozeilen-Optionen für `dprofpp`. Eine der interessantesten ist `-T`, mit der man einen Überblick darüber erhält, welche Subroutine jeweils von welcher anderen aufgerufen wird. Für weitere Optionen siehe `man dprofpp`.

Eine Anmerkung noch: die Ergebnisse sind umso genauer, je öfter eine Zeile / Subroutine durchlaufen wird. Bei einem oder wenigen Durchläufen kann es passieren, dass z.B. die verbrauchte Zeit sich bei `Devel::DProf` nicht einmal annäherungsweise auf 100 % addiert.

Analyse nach Zeilen

Normalerweise ist wohl die Effizienzanalyse nach Subroutinen interessanter als die nach einzelnen Zeilen. Sie kann aber trotzdem eine interessante Ergänzung sein, besonders für die Überprüfung der Effizienz von regulären Ausdrücken. Komplexe reguläre Ausdrücke können durchaus einen erhöhten Zeitaufwand für einzelne Zeilen verursachen, besonders wenn sie nicht mit einem festen Zeichen beginnen sondern schon am Anfang von mehreren Zeichen erfüllt werden.

Ein PERL-Modul zur zeilenweisen Effizienz-Analyse ist `Devel::SmallProf`. Das Wort Small darin bezieht sich laut manpage darauf, dass es bei der Abarbeitung des zu analysierenden Programms wenig

zusätzlichen Aufwand erzeugt. Bei einem Vergleichstest verzögerte es den Ablauf von obigem Testprogramm jedoch sehr viel stärker als `Devel::Dprof`. Im Gegensatz zu `Devel::Dprof` ist jedoch der Output (`smallprof.out`) menschenlesbar.

Im Output kommen zuerst eine Reihe von Analysen zu `strict` und `warnings`, bei der Analyse unseres Beispielprogramms geht man am besten direkt ans Ende der Datei. `count` sollte die Anzahl der Aufrufe der Zeile angeben, aber z.B. der Wert "2" für die Zeile `my @values = set_variables();` entzieht sich meiner Logik. Es folgt die "wall time". Das ist die Zeit, die laut einer Uhr an der Wand vergangen ist bei der Abarbeitung der Zeile (also die "Total Elapsed Time"); danach folgt die CPU-Zeit. Warum bei einer Reihe von Zeilen die "wall time" geringer ist als die "cpu time" entzieht sich ebenfalls meiner Logik. Bei der Messung beider Zeiten wird auf unterschiedliche Mechanismen zurückgegriffen und außerdem werden kleine Korrekturfaktoren eingebaut, um systemimmanente Ungenauigkeiten auszugleichen. Möglicherweise liegt darin der Grund für das merkwürdige Verhältnis der beiden Werte zueinander.

Auch wenn die Werte als solche somit wohl etwas zweifelhaft sind, sieht man dennoch sehr schnell, welche Zeilen besonders oft abgearbeitet werden und die meiste Zeit verbrauchen. Um einen schnellen Überblick über die am häufigsten ausgeführten Zeilen zu erhalten, kann man folgenden Befehl auf der Shell aufrufen:

```
sort -k1,1nr < smallprof.out | less
```

NYTProf

Es gibt inzwischen mit NYTProf noch einen besseren Profiler für PERL. Er muss über die CPAN-Shell installiert werden und kann dann wie folgt genutzt werden, um das Perl-Skript `test.perl` zu analysieren:

```
1 > perl -d:NYTProf test.perl
2 > nytprofhtml
3 > firefox nytprof/index.html
```

NYTProf erzeugt als Ausgabe die Datei `nytprof.out`, welche das Skript `nytprofhtml` in HTML-Dateien konvertiert. Diese können mit einem Browser angezeigt werden. NYTProf misst die Zeiten genauer und erzeugt eine komfortablere Ausgabe.

18.3 Zeiteffizientes Programmieren mit RE

Wer hier einen kompakten Überblick erwartet hat, wie durch Anwendung einiger Regeln die eigenen Programme schneller laufen, den muss ich wohl enttäuschen. In meinen Augen ist einer der wichtigsten Punkte, dass man sich vor dem Schreiben eines Programms erstmal hinsetzt, sich überlegt, welche Schritte man (von den groben bis dann hin zu den feineren) alle durchführen muss, und wie man sie sinnvoll anordnet. Damit erhöht man die Chancen deutlich, dass das Programm nachher eine klare Ablaufstruktur hat – und tendenziell auch schneller läuft als wenn man einfach drauflosschreibt. Nach Schreiben des Programms können dann die o.g. Tools wertvolle Hinweise darauf geben, welche Subroutinen besonders viel Zeit verbrauchen und wo man vielleicht umstrukturieren sollte.

Was konkrete PERL-Anweisungen betrifft, gibt es in Programming PERL, S. 593–603 eine ganze Reihe von eher unzusammenhängenden Tipps zur Beschleunigung. Für uns scheinen mir diejenigen besonders nützlich, die sich auf reguläre Ausdrücke beziehen (= "regular expressions" = RE), da wir die besonders häufig benutzen; deshalb soll dieses Thema näher beleuchtet werden.

Die Option "o" des Matching Operators

Wichtig zu wissen ist, dass reguläre Ausdrücke jedes mal neu kompiliert werden, sobald sie abgearbeitet werden. Das ist auch sinnvoll, denn sie können ja z.B. Variablen enthalten wie `/\b$word\b/`, und der Wert dieser Variablen muss jedesmal aktuell ermittelt werden. Oft werden reguläre Ausdrücke aber in Schleifen verwendet, wie z.B. in

```
1 while (<>) {
2     print "match!\n" if /interessant/;
3 }
```

Das ist recht zeitintensiv, da bei jedem Schleifendurchlauf der reguläre Ausdruck neu kompiliert wird. In Fällen wie diesem empfiehlt es sich sehr, die Option "o" des Matching Operators zu benutzen, also `/interessant/o`. Diese Option sorgt dafür, dass der reguläre Ausdruck nur beim ersten Mal kompiliert wird und in der Folge immer der schon kompilierte reguläre Ausdruck verwendet wird (das "o" steht für "once", d.h. "compile only once"). Im obigen Beispiel führt dies offensichtlich zu keinen Problemen, da sich der Suchstring ohnehin nie ändert.

Anders sieht es aus, wenn im Suchstring Variablen enthalten sind. Dort ist auch der Zeitaufwand für die Kompilation höher, weil immer der Wert der Variablen überprüft werden muss. Allerdings muss man sich genau überlegen, ob wirklich immer der Wert der Variablen, der beim ersten Erreichen des regulären Ausdrucks aktuell ist, auch für alle anderen Durchläufe gültig sein soll. In einem Fall wie

```
1 sub print_matches {
2
3     my $interesting_word = "activation";
4
5     while (<>) {
6         print "match!\n" if /$interesting_word/o;
7     }
8
9 }
```

kann man dies bedenkenlos tun (da hätte man auch den String direkt in den matching operator schreiben können), in einem Fall wie

```
1 sub print_matches {
2
3     my $interesting_word = shift;
4
5     while (<>) {
6         print "match!\n" if /$interesting_word/;
7     }
8
9 }
```

hingegen sollte man die Option sowieso vermeiden, da sonst bei jedem Aufruf der Subroutine nach demjenigen Wort gesucht würde, das beim ersten Aufruf der Subroutine als Argument mitgegeben wurde.

Eine nützliche Alternative: der qr// -Operator

Seit einigen Versionen gibt es in PERL einen Operator, der es uns sehr viel einfacher macht zu steuern, wann ein RE neu kompiliert werden soll und wann nicht – man kann jetzt REs kompilieren und kompiliert speichern ohne dabei zusätzlich noch eine Operation wie z.B. Matching durchzuführen:

```
1 $compiled_re = qr/$pattern/;
```

Schreibt man dann so etwas wie

```
1 foreach my $compiled_re (@compiled_res) {
2     $a++ if /$compiled_re/;
3 }
```

ist es wirklich nur noch das Aufrufen des gespeicherten REs, aber nicht mehr wie in

```
1 foreach my $interesting_word (@interesting_words) {
2     $a++ if /$interesting_word/;
3 }
```

das Abfragen einer Variablen, deren Inhalt dann erst in einen RE kompiliert werden muss.

study

Eine ganz andere Art der Effizienzoptimierung ermöglicht der PERL-Befehl `study`. Die Idee dabei ist, erstmal ein bisschen Zeit zu investieren, um einen String genauer zu analysieren, um danach schneller darauf suchen zu können. Dies ist natürlich besonders dann interessant, wenn man auf ein und demselben String mehrere Suchoperationen durchführen möchte.

Beispiel:

```
1 while (<>) {
2
3     study;  # bezieht sich immer auf $_, soweit nichts anderes
          angegeben wird
4     print "hallo\n" if /\bhallo\b/o;
5     print "interessant\n" if /\binteressant\b/o;
6     print "vielleicht\n" if /\bvielleicht\b/o;
7     ......
8
9 }
```

Während man also mit der Option "o" beim Suchmuster ansetzt, setzt man mit `study` beim String an, auf dem die Suche durchgeführt werden soll – und wie im Beispiel kann und sollte man ruhig beides kombinieren.

Ob `study` aber wirklich einen Zeitvorteil bringt, kann man nicht pauschal vorhersagen – immerhin verbraucht `study` selbst etwas Zeit. Im Zweifelsfall bleibt nur die Analyse mit einem der Tools zur Effizienzanalyse.

Wortsuche über Dateien hinweg

Im Folgenden soll auf möglichst effiziente Weise das Problem gelöst werden, dass auf verschiedenen Dateien eine ganze Liste von Wörtern gesucht werden soll – normalerweise eine ziemlich ineffiziente Sache.

```
1  #!/usr/bin/perl
2
3  # Programmname: suche_in_dateien.perl
4
5  use strict;
6  use warnings;
7
8  our @files = ("a.txt", "b.txt", "c.txt");
9  our @words = ("hallo", "wer", "da", "ist", "niemand");
10 our %patterns = ();
11 our %hits = ();
12
13 # Kompilieren der Suchmuster:
```

```
14 foreach my $word (@words) {
15     $patterns{$word} = qr/\b$word\b/;
16 }
17
18 @ARGV = @files; # Der angle-Operator (<>) soll automatisch die in
       @files enthaltenen Dateien nacheinander einlesen, so als waeren
       sie als Kommandozeilenargumente uebergeben worden.
19
20 undef $/; # jeweils ganzen Dateiinhalt auf einmal lesen
21
22 # Suche in den Dateien:
23 while (<>) {
24      study;
25     foreach my $word (keys%patterns) {
26         $hits{$ARGV}{$word}++ while /$patterns{$word}/g;
27     }
28 }
29
30 # Ausgabe der Treffer
31
32 foreach my $file (@files) {
33   print "Treffer in Datei $file:\n";
34   foreach my $word (sort keys%{$hits{$file}}) {
35      print "$word\t(", $hits{$file}{$word}, " mal)\n";
36   }
37   print "\n";
38 }
```

Bei diesem Vorgehen muss man natürlich sicherstellen, dass die gesuchten Wörter keine Metazeichen enthalten, die innerhalb eines regulären Ausdrucks eine besondere Bedeutung haben (*, +, ., (,), etc.). Das beschriebene Vorgehen lohnt sich umso mehr, je mehr Dateien durchgegangen werden sollen und umso mehr Wörter darin gesucht werden sollen.

Zahlreiche weitere Tipps zur Beschleunigung von Perlprogrammen finden sich in Programming PERL, S. 593-603.

19

Systemaufrufe

19.1 Vorbemerkung: Fehlerabfrage mit or

Die Operatoren `and` und `or` dienen zur Verknüpfung innerhalb von Boole'schen Ausdrücken. Bei `and` ist der Gesamtausdruck wahr, wenn die beiden verknüpften Aussagen wahr sind. PERL ist bei der Auswertung faul und bricht die Auswertung ab, wenn schon die erste Aussage nicht wahr ist, denn damit steht bereits fest, dass der Gesamtausdruck nur falsch sein kann.

Beispiel:

```
1  if (($a > 7) and ($b < 3)) {
2  ...
```

Stellt PERL fest, dass $a nicht größer ist als 7, dann wird die Auswertung abgebrochen, die Gesamtbedingung wird mit "falsch" bewertet.

Entsprechendes gilt für `or`. Hier ist die Gesamtaussage wahr, wenn mindestens eine der verknüpften Aussagen wahr ist. Ist bereits die erste Aussage wahr, steht das Gesamtergebnis bereits fest: auch die Gesamtaussage ist wahr. Wie bei `and` bricht PERL in diesem Fall die Auswertung ab:

```
1  if (($a > 7) or ($b < 3)) {
2  ...
```

Ist $a größer als 7, dann prüft PERL nicht mehr nach, ob $b kleiner als 3 ist oder nicht; die Gesamtbedingung wird als "wahr" bewertet.

Ausgewertet werden bei solchen logischen Verknüpfungen auch PERL-Kommandos bzw. deren Rückgabewerte. Dies macht man sich oft für bedingte Programmabbrüche zunutze, wie z.B. in

```
1  open FILE, "file.txt" or die "Couldn't open file: $!";
```

Die Verknüpfung mit `or` sorgt dafür, dass PERL versucht, den Boole'schen Wert der gesamten Zeile zu berechnen. Die `open`-Anweisung wird auf jeden Fall ausgewertet – und dazu, quasi als notwendiger Zwischenschritt – auch ausgeführt. Der Rückgabewert wird evaluiert, und in Abhängigkeit davon wird entschieden, ob auch der zweite Teil der Verknüpfung ausgewertet werden muss. Konnte der `open`-Befehl fehlerfrei durchgeführt werden, dann liefert er einen "wahren" Wert zurück, und PERL verzichtet auf die Auswertung (und dazu Durchführung) des zweiten Teils der Verknüpfung. Liefert `open` jedoch eine Null zurück, um ein Fehlschlagen des Befehls zu signalisieren, führt PERL auch den zweiten Teil der Verknüpfung aus, um den Boole'schen Wert der gesamten Zeile bewerten zu können – was in diesem Fall dazu führt, dass das Programm mit einer Fehlermeldung abgebrochen wird. Auf diese Weise wird eine logische Auswertung der Zeile dazu "missbraucht", den zweiten Teil in Abhängigkeit des Rückgabewertes des ersten Teils durchzuführen oder eben nicht. Das Ganze könnte man auch so schreiben:

```
1  if ((open FILE, "file.txt") == 0) {
2       die "Couldn't open file: $!";
3  }
```

19.2 Aufruf von anderen Programmen aus PERL heraus

Es gibt mehrere Möglichkeiten, aus PERL heraus andere Programme aufzurufen: `system()`, `exec()`, `open()` und Backticks.

Alle Möglichkeiten nutzen die Shell, um andere Programme zu starten (Ausnahme: `system()` und `exec()` benutzen die Shell genau dann nicht, wenn sie mit mehr als einem Argument aufgerufen werden). Die Hauptunterschiede zwischen den verschiedenen Möglichkeiten sind folgende:

system()

Bei `system()` wartet das PERL-Programm, bis die mit `system()` ausgeführte Anweisung beendet ist, danach läuft das PERL-Programm weiter. Der Output der ausgeführten Anweisung wird nicht gesammelt.

Beispiel:

```
1 ....
2 $daten = create_data();
3 system("mkdir daten");
4 open OUT, ">daten/neue_daten.txt" or die "$!";
5 print OUT $daten;
6 ....
```

Eine kleine Falle im Umgang mit dem `system`-Befehl besteht darin, dass er bei erfolgreicher Ausführung einen Nullwert zurückliefert und andernfalls einen Fehlercode. Dies steht im Gegensatz zu anderen PERL-Anweisungen, wie z.B. `open()`, die einen Nullwert gerade dann zurückgeben, wenn etwas schiefgelaufen ist.

Während man also bei `open()`, wie im obigen Beispiel, eine `or`-Verknüpfung nutzen kann, um auf Fehler zu reagieren (die `die`-Anweisung wird nur erreicht, wenn der Teil vor dem `or` im Boole'schen Sinne falsch ist, denn nur dann wird der Teil hinter dem `or` überhaupt ausgewertet), muss man bei `system()` zum Abfangen von Fehlern entweder eine `and`-Verknüpfung benutzen:

```
1 system("mkdir daten") and die "Couldn't create directory:
2                                $?";
```

oder aber einen Abgleich auf Null einbauen:

```
1 system("mkdir daten") == 0 or die "Couldn't create directory:
2                                $?";
```

Mit Fehlerabfrage würde obiges Beispielprogramm also wie folgt aussehen:

```
1  ....
2 $daten = create_data();
3 system("mkdir daten") == 0 or die "Couldn't create directory:
4                                $?";
5 open OUT, ">daten/neue_daten.txt" or die "$!";
6 print OUT $daten;
7 ....
```

Was Sicherheitsaspekte betrifft, ist die eingangs erwähnte Eigenschaft sehr wichtig, dass `system()` die Shell nur dann benutzt, wenn es mit genau einem Argument aufgerufen wird, wie z.B. im obigen `system("mkdir daten")` . Hierbei ist nicht entscheidend, wie viele Argumente im String aufgeführt sind, sondern dass dem `system`-Befehl als solchem insgesamt genau ein String übergeben wird (der so viele Argumente enthalten kann, wie er will). Um eine Nutzung der Shell zu verhindern, kann man also den `system`-Befehl einfach mit einer Liste von Strings aufrufen:

```
1 system("mkdir", "daten");
```

Warum es manchmal so wichtig ist, die Shell zu vermeiden, werden wir beim Thema Sicherheitsaspekte näher besprechen. Angemerkt sei noch, dass die Shell nicht zwangsläufig dieselbe ist, die man sich für seine Konsole-Fenster am Rechner ausgesucht hat. Es kann also durchaus sein, dass man normalerweise mit einer bash-Shell arbeitet, ein system-Kommando im PERL-Programm aber /bin/sh aufruft.

exec()

exec() funktioniert genau wie system() – mit dem wesentlichen Unterschied, dass das Perlprogramm nicht wartet, bis der Aufruf abgearbeitet ist. Stattdessen wird das Perlprogramm beendet. Eine Fehlerabfrage mit and oder or wie bei system() macht deshalb bei exec() keinen Sinn mehr – das Perlprogramm hätte vor der Bearbeitung der Fehlerkontrolle schon seinen Dienst eingestellt.

Genau wie bei system() kann eine Nutzung der Shell vermieden werden, indem mehrere Strings als Argumente übergeben werden.

Backticks

Bei den Backticks passiert weitgehend dasselbe wie bei system(), aber der Output wird gesammelt und steht im PERL-Programm als Rückgabewert zur Verfügung.

```
1 $current_directory = `pwd`;
```

Die Variable enthält dann das, was das Unix-Kommando sonst auf der Shell ausgegeben hätte. Erzeugt werden die Backticks auf der deutschen Tastatur als abwärts gerichtete Akzente über einem Leerzeichen.

Bei Backticks gibt es keine Variante des Aufrufs, um die Shell-Nutzung zu vermeiden. Die Backticks sollten nur genutzt werden, wenn auch tatsächlich der Rückgabewert benötigt wird; andernfalls ist ein system() effizienter (denn es bedeutet für PERL einen gewissen Aufwand, den Rückgabewert einzusammeln).

open()

open() wird vor allem in Verbindung mit Pipes zum Starten von Programmen verwendet.

Es soll hier nicht genauer auf alle Varianten und Details eingegangen werden, aber als Beispiel sei genannt:

```
1 open (TAGGER, "| cistagger > tagged_text.txt") or die;
2 print TAGGER $text;
3 close(TAGGER) or die;
```

Hier landet alles, was man in das Filehandle TAGGER reinschreibt, im Standard-Input des Tagger-Programms. Umgekehrt kann man aus Filehandles auch lesen:

```
1 open (PWD, "pwd |") or die;
2 $current_directory = <PWD>;
3 close (PWD) or die;
```

Dies wäre eine andere Variante, um das aktuelle Arbeitsverzeichnis abzufragen. Auch hier wird eventuell die Shell benutzt; neuere PERL-Versionen erlauben auch eine Art des Aufrufs, der die Shell-Nutzung ausschließt.

In beiden Beispielen wartet close() jeweils solange, bis das aufgerufene Programm endet.

20

Sicherheitsaspekte bei der Perlprogrammierung

Sicherheitsprobleme bei Sockets können auftreten, wenn ein Client (oder jemand, der sich per `telnet` mit dem Server-Socket verbindet) durch seine Eingaben unerwünschte Reaktionen des Servers bewirken kann wie z.B. Abstürzenlassen des Servers, Anzeigen von Dateiinhalten, Löschen von Dateien u.ä. Ein solches Sicherheitsproblem kann z.B. durch die Verwendung des `eval`-Befehls entstehen .

20.1 Exkurs: der `eval`-Befehl

Dem `eval`-Befehl können entweder ein String oder ein Anweisungsblock übergeben werden. Der Inhalt des Strings bzw. des Anweisungsblocks wird dann von `eval` als ein Stück PERL-Programm interpretiert und ausgeführt. Wir werden hier nur die Übergabe eines Strings näher betrachten.

Beispiel:

```
1  #!/usr/bin/perl -w
2
3  use strict;
4
5  our $c;
6
7  my $a = 3;
8  my $b = 7;
9  my $addition = '$c = $a + $b';
10 eval ($addition);
11 print "Result: $c\n";
```

Output:

```
Result: 10
```

Obwohl $addition nur ein String ist, wird das Ergebnis tatsächlich berechnet. Dies geschieht eben dadurch, dass `eval` den String als PERL-Code interpretiert und ausführt.

Nun ist die Berechnung der Summe von 2 Variablen kein Sicherheitsrisiko. Das Sicherheitsrisiko besteht darin, dass je nach Programmierung des Sockets jemand von außen darüber bestimmen kann, welche Strings mit `eval()` ausgewertet werden.

Verarbeitet `eval()` einen String, der "von außen" (z.B. als Eingabe in einer CGI-Seite, über `telnet`, über ein Client-Socket) in das Programm kommt, dann kann dieser Außenstehende praktisch beliebigen Programmcode auf dem Rechner des Sockets ausführen.

Beispiel für ein unsicheres Socket:

```
1  while (my $client = $server->accept()) {
2    # $client is the new connection
3
4    my $question = <$client>;
5    chomp($question);
6    print STDERR "Receiving:\t$question\n";
7
8    $question =~ /How much is (.*?)\?/io;
9    my $answer = eval($1);
10
11   print STDERR "Answering:\t$answer\n";
12   print $client "$answer\n";
13
14   close($client);
15   print STDERR "\nWaiting for the next connection ...\n\n";
16
17 }
18
19 close($server);
```

Gedacht war der Server für Anfragen, wie sie z.B. von folgendem Client gestellt werden:

```
1  foreach my $i (1..10) {
2    my $socket = IO::Socket::INET->new(PeerAddr => $remote_host,
3                                       PeerPort => $remote_port,
4                                       Proto    => "tcp",
5                                       Type     => SOCK_STREAM)
6      or die "Couldn't connect to $remote_host:$remote_port : $@\n";
7
8    print STDERR "Asking:\tHow much is $i * $i?\n";
9    print $socket "How much is $i * $i?\n";
10   my $answer = <$socket>;
11   print "Receiving:\t$answer\n";
12   close($socket);
13 }
```

Übergibt ein Client-Socket (oder ein per `telnet` mit dem Server-Socket verbundener Benutzer) dem Server-Socket eine Frage vom Muster

```
"How much is <programmcode>?",
```

so wird der Programmcode vom Server-Socket durch das `eval()` ausgeführt. Ein User kann jetzt eine Anfrage so stellen, dass das Server-Socket in eine Endlos-Schleife geführt wird:

```
"How much is 1 while(1)?"
```

Wird ein Systemaufruf an einen `eval`-Befehl übergeben, dann wird er ebenfalls ausgeführt – d.h. ein User kann beliebige Befehle auf der Shell aufrufen, bis hin zu `rm -rf *` und Ähnlichem.

Eine (noch sehr harmlose) Beispiel-Anfrage:

```
"How much is `ls`?"
```

Damit erhält der Außenstehende Einblick in den Inhalt des Verzeichnisses, in dem das Server-Socket läuft. Dieses Beispiel funktioniert nur mit Backticks, nicht z.B. mit `system()`, weil nur Backticks einen Rückgabewert erzeugen, und nur dieser wird als Antwort verschickt. Geht es aber nur um das Ausführen von Befehlen, so kann auch `system()` verwendet werden.

20.2 PERL's taint check

PERL hat einen eingebauten Modus, in dem grundsätzlich alle Variablen als "tainted" eingestuft werden, deren Daten von außerhalb des Programms eingelesen werden (*engl. to taint = verderben*, ich übersetze es hier mal als "manipuliert").

Der `taint`-Modus kann mit der Kommandozeilen-Option `-T` aktiviert werden. Daten, die als "tainted" eingestuft sind, dürfen in diesem Modus nicht verwendet werden innerhalb von Shell-Kommandos, Änderungen an Dateien, Verzeichnissen oder anderen Prozessen. Falls man dies doch versucht, bricht das Programm ab. Alle Variablen, in die (potenziell) manipulierte Variablen einfließen, gelten dabei automatisch ebenfalls als manipuliert.

Beispiele:

```
1 $arg = shift;              # $arg is tainted
2 $hid = "$arg, 'bar'";      # $hid is also tainted
3 $line = <>;                # tainted
```

Folgendes Programm z.B. bricht im "taint mode" während der Ausführung ab:

```
1 #!/usr/bin/perl -wT
2
3 use strict;
4
5 print "Bitte geben Sie ein Wort ein: ";
6 my $wort = <>;
7 open OUT, ">$wort.txt" or die;
8 print OUT $wort;
9 close OUT;
```

Output:

```
parker~/<2>security> test.perl
Bitte geben Sie ein Wort ein: hallo
Insecure dependency in open while running with -T switch at test.perl line 7, <> line 1.
```

Weitere Beispiele:

```
1 $mine = 'abc';             # Not tainted
2 $shout = `echo abc`;       # Tainted;
3 # die Shell kann mit sicheren Daten aufgerufen werden, aber das
      Ergebnis gilt als manipuliert.
4 $shout = `echo $shout`;    # Insecure (d.h. Programmabbruch)
5 # Shellaufruf mit einer manipulierten Variablen
```

Ein Beispiel von Manipulation:

Wird `` `echo abc` `` aufgerufen, macht das nichts weiter als "abc" auf dem Bildschirm auszugeben. Wird aber eine aus eine Benutzereingabe stammende, ungeprüfte Variable mit `` `echo $abc` `` ausgegeben, so könnte ein böswilliger Benutzer `abc;rm -f *` eingegeben haben – und alle Dateien im aktuellen Verzeichnis wären gelöscht:

```
1 #!/usr/bin/perl
2
3 use strict;
4 use warnings;
5
6 print "Bitte geben Sie ein Wort ein: ";
7 my $wort = <>;
8 print "Das eingegebene Wort lautete: ";
9 system "echo $wort";
```

Wirkung:

```
parker~/<3>test> test_2.perl
Bitte geben Sie ein Wort ein: abc;rm -f *
Das eingegebene Wort lautete: abc
parker~/<3>test> ll
insgesamt 8
drwxr-xr-x    2 hbosk    cisinter     4096 Jun 25 18:25 .
drwxr-xr-x    3 hbosk    cisinter     4096 Jun 25 18:25 ..
```

Wird `system()` mit mehreren Argumenten aufgerufen, so können auch manipulierte Daten verwendet werden, da die Shell nicht benutzt wird – und in Variablen verborgene Shell-Kommandos können nicht zur Ausführung kommen.

```
1  #!/usr/bin/perl -w
2
3  use strict;
4
5  print "Bitte geben Sie ein Wort ein: ";
6  my $wort = <>;
7  print "Das eingegebene Wort lautete: ";
8  system "/bin/echo", $wort;
```

Wirkung:

```
parker~/<3>test> test_2.perl
Bitte geben Sie ein Wort ein: abc;rm -f *
Das eingegebene Wort lautete: abc;rm -f *

parker~/<3>test> ll
insgesamt 12
drwxr-xr-x    2 hbosk    cisinter     4096 Jun 25 18:27 .
drwxr-xr-x    3 hbosk    cisinter     4096 Jun 25 18:27 ..
-rwxr--r--    1 hbosk    cisinter      155 Jun 25 18:27 test_2.perl
```

Auch einige Umgebungsvariablen gelten grundsätzlich als manipuliert, so z.B. `PATH`. Sie gelten erst als sicher, wenn sie explizit im Programm neu belegt wurden. Aus diesem Grund werden alle Shell-Aufrufe abgebrochen, die implizit auf `PATH` zurückgreifen.

```
1  system "echo $mine";   # Gilt als unsicher, falls $ENV{PATH} nicht
       neu belegt wurde
2  system "echo $hid";    # Doppelt unsicher, wenn $hid nicht sicher
       gemacht wurde
3
4  $path = $ENV{PATH};    # $path gilt als unsicher, da $ENV{PATH} ohne
       Neubelegung als unsicher gilt
5
6  $ENV{PATH} = '/bin:/usr/bin';  # jetzt gilt $ENV{PATH} als sicher
7  $ENV{IFS} = "" if $ENV{IFS} ne "";
8
9  $path = $ENV{PATH};      # $path jetzt auch sicher
10 system "echo $mine";      # Jetzt auch sicher
11 system "echo $hid";       # Immer noch unsicher, weil $hid unsicher
       ist.
```

Dateien, deren Name eine potenziell manipulierte Variable enthält, dürfen nur zum Lesen geöffnet werden:

```
1  open(OOF, "< $arg");      # gilt als OK (da nur zum Lesen geoeffnet
       wird)
2  open(OOF, "> $arg");      # unsicher, da Schreib-Zugriff
3
4  open(OOF, "echo $arg|"); # Unsicher weil $arg unsicher ist
5
6  $shout = `echo $arg`;     # Ebenso
7
8  unlink $mine, $arg;       # Und auch hier
9
10 exec "echo $arg";         # Unsicher, da bei einem einzigen String
       als Argument die Shell ins Spiel kommt
11 exec "echo", $arg;        # Gilt als sicher, weil die Shell nicht
       benutzt wird.
```

Die einzige offizielle Art, aus manipulierten Daten sichere Daten zu machen, ist die Verwendung von regulären Ausdrücken. Der Inhalt einer Variablen muss mit einem regulären Ausdruck abgeglichen werden, die back references gelten dann als sicher.

```
1  if ($addr =~ /^([-\@\w.]+)$/) {
2      $addr = $1;        # $addr gilt jetzt als sicher
3  }
4  else {
5      die "Potenziell gefaehrliche Zeichen in $addr";
6  }
```

$addr würde allerdings genauso als sicher gelten, wenn die erste Zeile

```
1  if ($addr =~ /^(.*?)$/) {
```

gelautet hätte – was $addr in Wirklichkeit kein bisschen sicherer macht. Sinn dieser Regelung ist es, dass der Programmierer die regulären Ausdrücke auch tatsächlich nutzt, um die Daten sicherer zu machen – die Verwendung des "taint mode" allein bewirkt das nicht.

Es gibt übrigens, rein technisch gesehen, auch noch eine andere Möglichkeit, manipulierte Daten in als sicher geltende zu verwandeln: die Verwendung als Schlüssel in einem Hash und das spätere Abfragen der Schlüssel. Auf diese Art können unbeabsichtigt Daten "sicher" gemacht werden.

Es gibt auch noch die Möglichkeit, die "Säuberung" von Daten durch reguläre Ausdrücke zu verhindern, um noch mehr Sicherheit zu gewinnen. Dazu wird das re-Pragma benutzt, das die Verwendung von regulären Ausdrücken kontrolliert:

```
1  use re 'taint';
2  ($dir, $file) = $fullpath =~ /(.*\/)(.*)/s;
```

So gelten $dir und $file auch nach der Benutzung eines regulären Ausdrucks noch als manipuliert, wenn $fullpath als manipuliert galt.

Soll dann eine Variable doch ganz bewusst bereinigt werden, so kann dies immer noch geschehen, in dem im aktuellen Block `re 'taint';` außer Kraft gesetzt wird:

```
1  use re 'taint';
2
3  {
4      no re 'taint';
5      if ($num =~ /^(\d+)$/) {
6          $num = $1;
7      }
8  }
9
10 # use re 'taint' gilt ansonsten im ganzen Programm.
```

20.3 Übungsaufgaben

Aufgabe 20.1 Schreibe das folgende Programm so um, dass es immer noch dasselbe tut, aber auch mit -T problemlos und sicher läuft:

```
1  #!/usr/bin/perl -w
2
3  use strict;
4
5  my $datei = shift;
6  die "Bitte Dateiname als Argument beim Aufruf eingeben!" unless
       $datei;
7  print STDERR "Bitte geben Sie ein Wort ein: ";
8  my $wort = <>;
9  print STDERR "Sie haben folgendes Wort eingegeben: ";
10 system "echo $wort";
11 print STDERR "Dieses Wort wird jetzt gespeichert.\n";
12 open OUT, ">$datei" or die "$!";
13 print OUT $wort, "\n";
14 close OUT;
```

Auch in der neuen Version soll der `echo`-Befehl verwendet werden.

Hinweise:

- Die Umgebungsvariable $PATH, im Programm ansprechbar als $ENVPATH, muss neu belegt werden. Für die Aufgabe reicht es, $ENVPATH auf `/bin` zu setzen (vorausgesetzt, dort befindet sich bei euch der `echo`-Befehl);
- Bitte beachten, dass das Sichern von Variablen geschieht, indem man der Variablen eine Backreference zuweist (z.B. `$variable = $1`), nachdem man die Variable gegen einen regulären Ausdruck gematcht hat.
- Es gibt 2 verschiedene Möglichkeiten, den `system`-Befehl sicher zu machen. Bitte im Programm BEIDE Möglichkeiten nutzen (auch wenn eine allein genügen würde).

21

CGI-Programmierung

21.1 Was ist CGI-Programmierung?

CGI-Programmierung wird zur Erzeugung von dynamischen Webseiten eingesetzt. Eine Webseite wird normalerweise vom Browser beim Server angefordert, der Server schickt die Seite an den Browser, und der stellt sie auf dem Bildschirm dar. Wenn es sich um eine statische Webseite handelt, nimmt der Server einfach die entsprechende Datei und schickt sie an den Browser. Bei dynamischen Webseiten hingegen gibt der Server die Anfrage des Browsers an ein Programm weiter, das Programm erzeugt einen HTML-Text und dieser wird dann an den Browser zurückgeschickt. Dieses Programm, das eine Anforderung (Request) an den Server entgegennimmt, und daraufhin eine Webseite erzeugt, nennt man ein CGI-Skript.

Dies kommt insbesondere zum Einsatz, wenn auf einer Webseite Formulare ausgefüllt werden und dann die eingegebenen Daten verarbeitet werden sollen und – in Abhängigkeit von den eingegebenen Daten – eine entsprechende nächste Webseite für den User generiert werden soll. Ein Beispiel: eine Webseite, auf der eine Online-Bestellung aufgegeben werden kann. Wird der "Submit"-Button (wie auch immer seine Beschriftung auf der Webseite lautet) angeklickt, schickt der Browser eine Request inkl. der Daten an den Server, und der leitet sie an das CGI-Skript weiter, das in der Request genannt wird. Das Skript verarbeitet dann die Daten und generiert eine neue Webseite, in der – je nachdem – auf fehlende Eingabedaten aufmerksam gemacht, eine Fehlermeldung ausgegeben oder die erfolgreiche Bestellung bestätigt wird.

Ein CGI-Skript kann in den verschiedensten Programmiersprachen geschrieben werden – es muss sich nur an ein paar vorgegebene Konventionen halten, z.B. wie die Eingabedaten aus der Request an das Programm übergeben werden und wie der Output formal auszusehen hat. Dieses Protokoll heißt CGI, daher der Name CGI-Programmierung. CGI ist die Abkürzung für "Common Gateway Interface".

Wo lege ich meine Webseiten und CGI-Programme ab?

Im CIP-Pool legt man für die Webseiten ein Unterverzeichnis "public_html" in seinem Homeverzeichnis an. Das Verzeichnis "cgi-bin" für CGI-Skripte legt man dann als Unterverzeichnis des Verzeichnisses "public_html" an. Allerdings registriert der Webserver immer erst über Nacht, dass ein User ein neues solches Verzeichnis angelegt hat, d.h. es kann erst ab dem folgenden Tag benutzt werden.

Ein erstes CGI-Programm

Wir werden zunächst ein kleines CGI-Programm schreiben, das nur eine Webseite generiert, ohne irgendwelche Daten aus einem Formular zu verarbeiten o.ä.

Ein solches Programm kann man im Prinzip ohne jegliche Module schreiben – man muss nur korrekte Header und eine gültige HTML-Seite auf `STDOUT` ausgeben. Das CGI-Modul von PERL erleichtert die Sache aber sehr. *Per default* importiert man die objektorientierten Methoden, aber wenn

man `:standard` als Argument beim Import angibt, so hat man die ganze Funktionalität auf nicht-objektorientierte Weise zur Verfügung:

```
1 use strict;
2 use warnings;
3 use CGI (":standard");
4
5 print header();
6 print start_html("Titel der Seite");
7 print p("Dies ist der Text der Seite");
8 print end_html();
```

Statt `p("Text")` kann man genauso gut direkt HTML schreiben: `print "<p>Text</p>";` – die Ausgabe ist in beiden Fällen die gleiche, und man kann in einem CGI-Skript problemlos beide Schreibweisen nebeneinander verwenden.

Viele HTML-Tags lassen sich so produzieren wie beim `p`-Befehl: der Befehl lautet wie das jeweilige Tag, und was in den Klammern als Argument mitgegeben wird, wird von einem öffnenden und einem schließenden Tag dieses Typs umschlossen. Vorsichtig sein sollte man nur, wenn man das `<tr>`-Tag erzeugen will (für Tabellenzeilen, Abkürzung von "table row"): da es `tr()` schon als PERL-Befehl gibt, lautet der Befehl für die Erzeugung der Tags `Tr()` und nicht "tr". (für den `Tr`-Befehl und andere Tabellenbefehle muss zusätzlich `:html3` importiert werden).

Aufruf selbstgeschriebener CGI-Skripte über einen Browser

Unter welcher Webadresse die abgelegten CGI-Skripte aufgerufen werden können, hängt vom jeweiligen Provider ab. Für das Institut der Informatik finden sich die genauen Angaben unter Homepages für Studenten.

Achtung: Zur Zeit (SS 2014) funktioniert der normale CGI-Server des IfI nicht richtig, und es sollte stattdessen der Testserver cgi2 angesprochen werden. Die URL lautet dann beispielsweise `http://cgi2.cip.ifi.lmu.de/~MeinLoginName/meinCGIskript.pl`.

Fehlersuche in CGI-Skripten

Wird ein fehlerhaftes CGI-Skript mit einem Browser aufgerufen, dann werden 2 Fehlermeldungen erzeugt:

1. Der Browser zeigt einen "500 Internal Server Error" an.
2. Die entsprechende Fehlermeldung von PERL wird in ein Error-Logfile des Servers geschrieben, auf dem das CGI-Skript läuft.

Das Dumme ist nur, dass einem die Browsermeldung mangels Genauigkeit nicht weiterhilft und nur Superuser das Error-Logfile des Servers lesen können.

Es gibt 3 Lösungsmöglichkeiten:

1. Man ruft das CGI-Skript auf der Shell auf. Das funktioniert in unserem Fall ganz gut, weil wir keine Formulardaten verarbeiten. Wenn das Programm allerdings Formulardaten verarbeitet, müssen sie über die Shell eingegeben werden, was auf Dauer zum Debuggen ziemlich lästig wird.
2. Man benutzt das Modul CGI::Carp und importiert `fatalsToBrowser`:

   ```
   1 use CGI::Carp qw(fatalsToBrowser);
   ```

Dann erscheint für jeden Fehler, der zum Programmabbruch führt, die PERL-Fehlermeldung im Browserfenster.

Das ist zwar deutlich hilfreicher zum Debuggen als ein "500 Internal Server Error" – aber man sieht keine Warnmeldungen und jeder kann weltweit mitlesen, was man genau für Fehler produziert.

3. Man benutzt das Modul `CGI::Carp`, importiert `carpout` und schreibt in ein eigenes Error-Logfile:

```
1  BEGIN {
2      use CGI::Carp qw(carpout);
3      open (LOG, ">>error.log") or die "unable to append message
          to the error logfile: $!";
4      carpout(*LOG);
5  }
```

Diese Variante ist wahrscheinlich für die meisten Fälle die empfehlenswerteste. (b) und (c) lassen sich auch kombinieren; man muss dann in (c) statt `qw(carpout)` `qw(carpout fatalsToBrowser)` schreiben.

Rechte des Skripts

CGI-Skripte laufen bei den Informatikern mit denselben Rechten, die auch der User hat, dem das Skript gehört. Damit kann das Programm genau dann z.B. Dateien anlegen / lesen / löschen, wenn der Programmbesitzer dies auch machen könnte. Dies ist allerdings nicht auf allen Servern so, wo man CGI-Skripte laufen lassen kann. Deshalb kann es sein, dass auf anderen Servern schon der Versuch scheitert, überhaupt das Skript zu starten oder ein Error-Logfile anzulegen, weil die Rechte des Programms nicht dazu ausreichen.

Tabellen erzeugen

Als nächste Übung soll eine Webseite mit einer Tabelle erzeugt werden. Tabellen werden in HTML durch das Tag `<table>...</table>` markiert. Dabei kann die Breite der Trennlinie beim öffnenden Tag angegeben werden, z.B. `<table border="1">`. Innerhalb der Tabelle werden Zeilen mit `<tr>...</tr>` umschlossen; die einzelnen Zellen der Tabelle werden entweder von `<th>...</th>` ("table header", üblicherweise falls die erste Zeile Spaltenüberschriften enthalten soll) umschlossen oder von `<td>...</td>` ("table data").

Um für Tabellenelemente auch entsprechende Funktionen zur Verfügung zu haben, muss man beim Aufruf des CGI-Moduls zusätzlich zu `:standard` auch `:html3` importieren.

Hinweis: SelfHTML[4] ist eine informative und praxisbezogene Webseite rund um HTML, die weitergehende Fragen beantwortet.

21.2 Aufruf eines CGI-Skripts

Ein CGI-Skript kann aus einer HTML-Webseite heraus auf verschiedene Arten aufgerufen werden:

- über ein Link;
- über ein Formular;

[4] `http://de.selfhtml.org/`

- über eine Server Side Include-Anweisung (um dynamisch erzeugten Text in eine ansonsten statische Webseite einzufügen);
- über ein Refresh, das als URL diejenige des CGI-Skripts enthält.

Um die Übergabe von Daten zu zeigen, soll im folgenden eine Formularseite erstellt werden, deren Daten dann von einem CGI-Skript verwertet werden.

21.3 Erstellen einer Formularseite

Formulare werden in HTML-Seiten von dem Tag `<form>...</form>` umschlossen. Mit dem Attribut `action` wird dabei angegeben, welches CGI-Skript beim Abschicken des Formulars aufgerufen werden soll, und mit dem Attribut `method` wird angegeben, mittels welcher Methode die Daten übergeben werden sollen.

```
<form action="/~MeinLoginName/cgi-bin/comments.pl" method="get">
....
</form>
```

Zwischen dem öffnenden und dem schließenden form-Tag können dann verschiedene Typen von Formularelementen eingebunden werden, z.B. einzeilige oder mehrzeilige Eingabefelder, Auswahllisten o.a.. Ein Tag `<input>` mit dem Attribut `type = "submit"` ermöglicht das Abschicken des Formulars, wobei das Attribut value angibt, was als Text auf dem Button erscheinen soll.

```
<html>
<head>
<title>Kommentarseite</title>
</head>
<body>
<h1>Ihr Kommentar</h1>
<form action="/~MeinLoginName/cgi-bin/comments.pl" method="get">
<p>Name:<br><input size="40" maxlength="40" name="AnwenderName"></p>
<p>Text:<br><textarea rows="5" cols="50" name="Kommentartext"></textarea></p>
<p><input type="submit" value="Absenden"></p>
</form>
</body>
</html>
```

Die Webseite, die dabei herauskommt, kann man sich anzeigen lassen, indem man den HTML-Code in einer Datei speichert und diese Datei mit dem Browser lädt.

Auswahllisten

Das Tag für Auswahllisten lautet `<select>`. Als Attribut wird wie bei `input` oder `textarea` ein Name angegeben. Zwischen `<select name="wasauchimmer">` und `</select>` werden dann die Optionen aufgelistet, aus denen der Benutzer auswählen kann.

```
<form action="/~MeinLoginName/cgi-bin/top10.pl" method="get">
<select name="Interpret">
<option>Heino</option>
<option>Michael Jackson</option>
<option>Tom Waits</option>
<option>Nina Hagen</option>
<option>Marianne Rosenberg</option>
</select>
<p><input type="submit" value="Absenden"></p>
</form>
```

21.4 Übergabe der Daten

Das CGI-Skript sieht 3 Methoden für die Kommunikation vor:

1. HEAD
2. GET
3. POST

Die Methode `HEAD` ist nur interessant, wenn man lediglich die Header einer Seite, aber nicht die Seite selbst abrufen will. Sie kommt deshalb für uns hier nicht in Frage.

`POST` ist zur Übergabe von Daten gedacht; diese werden dann bei der Kommunikation explizit als solche übergeben. `GET` ist eigentlich für die Anforderung von Dokumenten gedacht. Allerdings ist es möglich, auch bei `GET` Daten zu übergeben, und dies wird auch häufig benutzt. Die Daten müssen dann als Teil der Ziel-URL kodiert werden, wie z.B. in `http://dict.leo.org/?search=soccer&searchLoc=0&relink=on&deStem=standard&lang=en`.

Ein wichtiger Unterschied zwischen beiden Methoden ist, dass bei `GET` davon ausgegangen wird, dass die Anzahl der Requests keinen Unterschied machen soll – solange der Request (d.h. hier die URL) gleich ist, soll auch das Ergebnis immer dasselbe sein. Bei `POST` hingegen wird davon ausgegangen, dass jeder Request ein anderes Ergebnis produzieren kann. Wird z.B. ein Bestellformular von einem Online-Shop mit `POST` abgeschickt, so wird jede Bestellung so oft registriert, wie entsprechende Requests abgeschickt werden, auch wenn die Requests identisch sind. Eine andere sinnvolle Anwendung von `POST` wäre z.B. ein Formular für eine Meinungsumfrage, wo auch bei identischen Requests unbedingt jede Instanz erfasst werden soll. `GET` hingegen ist z.B. für Wörterbuchabfragen geeignet – dieselbe Abfrage soll immer dasselbe Resultat zurückliefern, die Anzahl der Abfragen hat auf der Serverseite keine inhaltlichen Auswirkungen. `GET`-Anfragen können deshalb z.B. auch im Cache von Proxies gespeichert werden, so dass man bei Mehrfach-Anfragen die Antwort von dort bekommt statt vom Server selbst. `POST`-Anfragen hingegen werden grundsätzlich nicht von einem Proxy bearbeitet, sondern immer an den Server weitergeleitet.

Ein Vorteil von `GET` ist, dass man als User ein Link oder ein Bookmark auf eine URL setzen kann, in der die Daten schon enthalten sind. Nachteilig ist `GET`, wenn der Skriptschreiber nicht will, dass die übergebenen Daten einfach sichtbar sind. Außerdem kann man auch nicht beliebig große Datenmengen auf diesem Weg übergeben.

21.5 Das Einlesen von Daten im CGI-Skript

Ohne das CGI-Modul ist das Einlesen der übergebenen Daten eine recht umständliche Sache – die Attribute und Werte müssen separiert werden und eine Reihe von Zeichen müssen dekodiert werden. Das CGI-Modul stellt jedoch eine sehr praktische Funktion namens `param()` zur Verfügung: Heißt auf einer Formularseite ein Eingabefeld z.B. "Vorname", so liefert mir die Zeile

```
1 my $value = param('Vorname');
```

den entsprechenden Wert und ich kann ihn weiterverarbeiten.

Ein CGI-Skript, das die Daten des Beispielformulars weiterverarbeitet, kann z.B. so aussehen:

```
1 #!/usr/bin/perl
2
3 use strict;
4 use warnings;
```

```
 5 use CGI qw(:standard escapeHTML);
 6
 7 BEGIN {
 8   use CGI::Carp qw(fatalsToBrowser carpout);
 9   open (LOG, ">>errors.log") or die
10                         "couldn't open log file: $!";
11   carpout(*LOG);
12 }
13
14 my $user = param('AnwenderName');
15 my $comment = param('Kommentartext');
16
17 print header();
18 print start_html("Kommentar-Echo");
19 print p("$user schrieb:");
20 print p(escapeHTML($comment));
21 print end_html();
```

Um Daten aus der weiter oben als Beispiel verwendeten Auswahlliste weiter zu verarbeiten, würde man `param("Interpret")` abfragen.

21.6 Die Ausgabe von Daten im CGI-Skript

Damit Text in korrektem HTML ausgegeben wird (inklusive der Named Entities, wie z.B. ü für "ü") kann die Funktion `escapeHTML()` des CGI-Moduls verwendet werden (siehe auch im Beispiel). Diese Funktion muss explizit importiert werden; sie ist nicht in `:standard` enthalten.

21.7 Erzeugen von Formularen mit einem CGI-Skript

Oft kommt es vor, dass man nicht nur von einer HTML-Seite aus ein CGI-Skript aufrufen möchte, das dann auf die Daten reagiert, die in einem Formular der HTML-Seite eingegeben wurden. Stattdessen möchte man, dass das CGI-Skript selbst ein Formular ausgibt, dessen Daten dann wieder von einem anderen oder auch vom selben CGI-Skript verarbeitet werden.

Das CGI-Modul bietet für die Erzeugung des öffnenden und schließenden `form`-Tags die beiden Anweisungen

1. `start_form()`
2. `end_form()`

Gibt man keinerlei Argumente an, dann bedeutet das automatisch, dass zur Bearbeitung der eingegebenen Daten das Skript selbst aufgerufen wird. Will man eine andere Aktion beim Absenden des Formulars auslösen, so kann dies wie bei den meisten anderen Befehlen des CGI-Moduls auch in folgender Form geschehen:

```
1 start_form(-action=>'/cgi-bin/script.pl');
```

Die Formularelemente selbst können mit den folgenden Anweisung erzeugt werden:

1. `textfield()`
2. `filefield()`
3. `password_field()`
4. `button()`

5. `checkbox()`
6. `checkbox_group()`
7. `radio_group()`
8. `image_button()`
9. `submit()`
10. `reset()`

Auch hier können die Tag-Attribute in der Form übergeben werden wie in dem Beispiel zu `start_form`. Eine Eingabezeile kann z.B. wie folgt erzeugt werden:

```
1 print textfield(-name=>'Vorname', -size=>50);
```

Ein Formular mit Eingabefenster und Submit-Button kann man also z.B. wie folgt in einem CGI-Skript erzeugen:

```
1 print start_form();
2 # da keine andere Aktion festgelegt ist, ruft sich das CGI-Skript
      zur Verarbeitung der Daten selbst auf
3
4 print textfield(-name=>'Kundennummer', -size=>10);
5
6 print submit(-value=>'Abschicken');
7 # Der value bestimmt, was auf den Submit-Button geschrieben wird
8
9 print end_form();
```

Ein komplettes CGI-Skript, das, falls noch keine Kundennummer eingegeben wurde, die Kundennummer per Formular abfragt, oder andernfalls die Kundennummer ausgibt, könnte dann wie folgt aussehen:

```
1  #!/usr/bin/perl -w
2
3  # get error messages in own logfile:
4
5  BEGIN {
6      use CGI::Carp qw(carpout);
7      open (LOG, ">>error.log") or die "could not append to
8                              error.log: $!\n";
9      carpout(*LOG);
10 }
11
12 use strict;
13 use CGI qw(:standard escapeHTML :html3);
14
15 print header();
16 print start_html('Ihre Kundennumer');
17
18 if(not(defined(param('Kundennummer')))) {
19     print start_form();
20     print "Bitte geben Sie ihre Kundennummer ein: ";
21     print textfield (-name=>'Kundennummer', -size=>20);
22     print submit(-value=>'Abschicken');
23     print end_form();
24 } else {
25     print p("Ihre Kundennummer lautet ", escapeHTML(param('
          Kundennummer')), ".");
```

```
26 }
27
28 print end_html();
```

21.8 Übungsaufgaben

Aufgabe 21.1 Schreibe ein CGI-Skript, das folgende Datenstruktur enthält und daraus eine Webseite mit einer entsprechenden Tabelle erzeugt:

```
1 my @books = (
2   {author => 'Wall, Larry', title => 'Programming Perl', year =>
       2001},
3   {author => 'Christiansen, Tom', title => 'Perl Cookbook', year =>
        1998},
4   {author => 'Stein, Lincoln', title => 'Official guide to
       programming with CGI.pm', year => 1998},
5   {author => 'Cross, David', title => 'Data Munging with Perl',
       year => 2001},
6   {author => 'Deitel, H.M.', title => 'Como programar en C/C++',
       year => 1994}
7 );
```

Aufgabe 21.2 Schreibe eine Webseite, die den Benutzer nach einem Sortierkriterium für Bücher fragt (Autor, Titel oder Jahr). Sobald der Benutzer auf den Button "Absenden" klickt, soll er zu der CGI-Seite aus Ausgabe 21.1 gelangen. Diese soll dann die Bücher ausgeben, diesmal aber sortiert nach dem vom Benutzer ausgewählten Sortierkriterium.

Aufgabe 21.3 Schreibe die Lösung zu Aufgabe 21.2 so um, dass nun ein CGI-Skript sowohl das Formular generiert, das das Sortierkriterium abfragt, als auch die sortierte Liste der Bücher anzeigt, sobald ein Sortierkriterium vorliegt.

22

PERL und Datenbanken

22.1 DBM-Dateien

Manchmal hat man große Datenmengen zu verwalten, die sich gut in einem Hash halten lassen – aber man will sie auch zwischen den Programmaufrufen nicht verlieren, und es dauert zu lange, die Datenmengen jedes Mal in eine Textdatei rauszuschreiben und beim nächsten Mal wieder aus der Textdatei in ein Hash zu lesen. Ein relativ typischer Fall, wo diese Situation auftritt, sind z.B. CGI-Programme, die schnell starten sollen, aber andererseits auch auf große Datenmengen zugreifen sollen.

Eine Lösung für diese Problematik bieten sog. DBM-Dateien. Die Grundidee ist die, dass man den aktuellen Inhalt eines Hashes in jedem Moment in einer speziellen Datei abbildet. Jedes Mal, wenn man im Programm etwas am Hash-Inhalt verändert, wird es auch in der zugehörigen Datei entsprechend geändert. Wenn man dann das Programm beendet, "entkoppelt" man die Datei vom Hash im Programm, so dass die Datei erhalten bleibt. Beim nächsten Programmaufruf koppelt man wieder die Datei an ein Hash – und hat so mit einem Schlag alle Daten im Hash verfügbar.

Der Nachteil der Methode ist, dass es natürlich mehr Aufwand ist, bei jeder Veränderung des Hash-Inhaltes auch die Datei entsprechend anzupassen. Daher lohnen sich DBM-Dateien nur für Anwendungen, die große Hashes zwischen verschiedenen Programmaufrufen speichern und dann schnell wieder laden sollen – wobei sich am Inhalt des Hashes nur wenig oder nichts ändert.

In einem speziellen Punkt unterscheiden sich DBM-Dateien allerdings von normalen Hashes: `exists()` und `defined()` sind in diesem Fall unterschiedslos. (wohingegen bei normalen Hashes Einträge mit dem Wert `undef` bei einer `exists`-Abfrage einen logisch wahren Wert zurückgeben, bei einer `defined` -Abfrage einen logisch falschen).

Es gibt verschiedene Implementierungen von DBM-Dateien; einige sind schneller im Zugriff, verbrauchen aber mehr Platz auf der Festplatte, andere umgekehrt etc. Wir werden hier die Berkeley-Implementierung benutzen.

22.2 Benutzung von DBM-Dateien unter PERL

Es gibt zwei Methoden, DBM-Dateien unter Perl zu benutzen. Die ältere von beiden Methoden ist das Öffnen mit `dbmopen()`; die neuere das Anbinden mit `tie()`. Bei neueren PERL-Versionen verbirgt sich hinter `dbmopen()` in Wirklichkeit auch nur ein Aufruf von `tie()`; wir werden hier deshalb direkt die `tie()`-Aufrufe besprechen.

```
1 #!/usr/bin/perl
2
3 use strict;
4 use warnings;
5
6 use DB_File; # wir waehlen die Berkeley-Implementierung
```

```
 7
 8  my $dbm_file = "daten.dbm";
 9  my %my_hash = ();
10
11  tie %my_hash, "DB_File", $dbm_file or
12                                  die "Can't open $dbm_file: $!";
13  # Argumente bei diesem Aufruf: das Hash, die gewaehlte DBM-
        Implementierung und der Name der Datei, an die das Hash
        gekoppelt werden soll
14  # Wenn die Datei noch nicht existiert, wird sie angelegt
15
16  $my_hash{dummy}++;
17  print STDERR "Der Zaehler fuer Dummy ist jetzt bei ", $my_hash{
        dummy}, "\n";
18  # und beliebige weitere Hash-Operationen
19
20  untie %my_hash;
```

Da `tie()` ein generelles Konzept ist, hängen die Argumente nach dem zweiten Argument davon ab, was konkret mit `tie()` angebunden wird. Im Fall eines `DB_File` können Zugriffsmodus und Rechte mitgegeben werden, müssen es aber nicht.

Enthält das Hash im Moment des tie-Aufrufs schon irgendwelche Werte, dann sind diese nach dem tie nicht mehr sichtbar, es enthält dann nur noch genau diejenigen Daten, die in der DBM-Datei abgelegt waren. Nach einem `untie()` enthält das Hash dann genau wieder diejenigen Werte, die vor dem tie darin enthalten waren.

22.3 Speichern von komplexen Datenstrukturen in DBMs

Was macht man, wenn man z.B. ein Hash of Hashes in einer DBM-Datei speichern will? Mit den o.g. Methoden ist das nicht möglich. Aber wieder mal gibt es ein entsprechendes PERL-Modul ... `MLDBM` in diesem Fall. Als Argument beim Einbinden gibt man die DBM-Implementierung mit, die man benutzen will:

```
1  use MLDBM 'DB_File';
```

Danach kann man dann auch komplexe Datenstrukturen verwenden:

```
1  $participants{"Perlkurs1"} = ["Hans", "Sabine", "Olaf", "Johannes"
       ];
```

Was man jedoch nicht machen kann, ist, einzelne Komponenten der Datenstruktur direkt verändern – man kann immer nur den komplexen Wert, der einem Schlüssel zugeordnet ist, als Ganzes abfragen oder abspeichern.

```
1  $participants{"Perlkur2"}->[0] = "Barbara"; # funktioniert nicht!
```

Stattdessen muss man sich die Referenz geben lassen (d.h. den Hash-Wert), die Komponente verändern, und dann die Referenz auf die veränderten Daten wieder dem Hash-Schlüssel zuweisen.

```
1  #!/usr/bin/perl
2
3  use strict;
4  use warnings;
5
6  use Fcntl qw(:DEFAULT);
7  use MLDBM 'DB_File';
8
9  {
10   our %teilnehmer;
```

```
11
12  my $dbm_file = "daten.dbm";
13
14  tie (%teilnehmer, 'MLDBM', $dbm_file, O_CREAT | O_RDWR, 0666)
15                      or die "couldn't tie $dbm_file: $!";
16
17  # komplexe Werte koennen zugewiesen werden:
18  $teilnehmer{"Perlkurs1"} = ["Hans", "Sabine", "Olaf", "Johannes"
        ];
19  $teilnehmer{"Perlkurs2"} = ["Norbert", "Elisabeth", "Ralf"];
20
21  teilnehmer_ausgeben("Perlkurs2");
22
23  # aber ich kann nicht einzelne Komponenten eines Wertes direkt
        manipulieren:
24  # $teilnehmer{"Perlkur2"}->[0] = "Barbara"; # funktioniert nicht!
25
26  # stattdessen: Referenz geben lassen, Komponente veraendern, neue
        Referenz abspeichern:
27
28  my $teilnehmer_perl2 = $teilnehmer{Perlkurs2}; # ich bekomme eine
        Referenz auf das Array
29  $teilnehmer_perl2->[0] = "Barbara"; # Komponente veraendern
30  $teilnehmer{Perlkurs2} = $teilnehmer_perl2; # Referenz wieder dem
        Key zuweisen
31
32  teilnehmer_ausgeben("Perlkurs2");
33
34  untie %teilnehmer;
35 }
36
37 sub teilnehmer_ausgeben {
38
39   our %teilnehmer;
40   my $kurs = $_[0];
41
42   print "Folgende Personen nehmen an $kurs teil:\n";
43   foreach my $teilnehmer (@{$teilnehmer{$kurs}}) {
44       print "$teilnehmer\n";
45   }
46   print "\n";
47 }
```

22.4 File Locking bei DBM-Dateien

Die Möglichkeiten des File Locking hängen sehr von der Implementation ab. Einige Implementationen, wie z.B. `SDBM` oder `GDBM`, haben ein eigenes File Locking – System; bei `DB_File` z.B. ist dies nicht der Fall, und man muss selbst das Locking organisieren. Zwei verschiedene Lösungen für diesen Fall finden sich in "Programming PERL" und im "PERL Cookbook".

22.5 MySQL-Nutzung

Einrichten einer Datenbank am Institut für Informatik

Wenn man Daten dauerhaft speichern und verwalten will, die sich nicht mehr so leicht in ein Hash fassen lassen und / oder die man während des Programms häufiger ändern will, dann empfiehlt sich

statt der Nutzung von DBM-Dateien die Einrichtung einer wirklichen Datenbank, z.B. MySQL. Eine solche Datenbank können sich Personen mit einem account bei den Informatikern unter `https://tools.rz.ifi.lmu.de/cipconf/` selbständig einrichten. Im Folgenden wird vorausgesetzt, dass das geschehen ist; wir werden jetzt die wichtigsten Kommandos besprechen, wie man so eine Datenbank benutzt.

Nutzung des mysql-Client-Programms

Von der Shell aus kann man mit

```
mysql -h db2.cip.ifi.lmu.de -u <login> -p <Name der Datenbank>
```

ein Client-Programm aufrufen, das beim Start eine Verbindung zur eingerichteten Datenbank aufbaut und mit dem man diese interaktiv konsultieren kann. Dieses Konsultieren erfolgt mit SQL-Anweisungen (SQL = "Structured Query Language"). Eine solche Anweisung muss dabei normalerweise mit einem Semikolon abgeschlossen werden. Zum Beenden des Clients gibt man `exit` oder `quit` ein, für Hilfe `help` oder `?` (diese Client-Kommandos brauchen nicht mit einem Semikolon abgeschlossen zu werden).

Eine erste Abfrage könnte dann so aussehen:

```
select version();
```

Daraufhin sollte so etwas am Bildschirm erscheinen:

```
+-------------+
 | version()   |
+-------------+
| 3.23.49-log |
+-------------+
1 row in set (0.00 sec)
```

Man kann auch mehrere Kommandos hintereinander eingeben:

```
select version(); select now();
```

oder sie auch zusammenfassen:

```
select version(), now();
```

Im letzteren Fall sollte die Ausgabe ungefähr so aussehen:

```
+-------------+---------------------+
| version()   | now()               |
+-------------+---------------------+
| 3.23.49-log | 2003-06-20 11:28:56 |
+-------------+---------------------+
1 row in set (0.00 sec)
```

Wäre das jetzt eine Abfrage einer tatsächlichen Datenbank, würden in der ersten Zeile die Namen der abgefragten Spalten aus der Datenbank stehen.

Klein- oder Großschreibung ist bei den Anfragen übrigens egal; wenn man das abschließende Semikolon vergessen hat, erscheint ein "->" in der nächsten Zeile, das weitere Teile der Anfrage erwartet. Dort kann man dann das fehlende Semikolon noch eingeben und die Anfrage abschicken.

Dies sollte nur eine erste Annäherung an die Benutzung der Datenbank sein; für alles weitere soll jetzt die Nutzung mit PERL im Vordergrund stehen. Wer sich näher mit dem mysql-Client befassen

will, findet ein gutes Tutorial unter `http://dev.mysql.com/doc/refman/6.0/en/tutorial.html` (in Englisch).

22.6 Nutzung einer MySQL-Datenbank mit PERL

Herstellen der Verbindung zur Datenbank

Für die Nutzung einer Datenbank aus PERL heraus gibt es das Modul `DBI` (DataBase Interface). Es fragt die Datenbank nicht direkt ab, sondern gibt die Anfragen nur an ein entsprechendes Treiberprogramm weiter, in unserem Fall `DBD::mysql`, wobei `DBD` für "DataBase Driver" steht). Informationsquellen für die Benutzung sind also `perldoc DBI` und in unserem Fall `perldoc DBD::mysql`.

Selbst in sein Programm einbinden muss man nur das DBI-Modul; die Einbindung des Treiber-Moduls wird dann vom DBI-Modul übernommen.

Der nächste Schritt nach `use DBI;` ist dann normalerweise, die Verbindung zu der Datenbank auf dem Datenbank-Server herzustellen. Dazu dient das Kommando `DBI->connect()`, dem man folgende Argumente mitgeben muss:

1. Einen String, der aus 3 mit Doppelpunkten getrennten Teilen bestehen muss:
   - `DBI`
   - Dem Namen des Treibers (ohne `DBD::`), in unserem Fall `mysql`
   - den Daten, die der Treiber für das Verbinden braucht. Für den mysql-Treiber ist das: `database=$database;host=$host`. In `$database` muss der Datenbankname stehen, in `$host` in unserem Fall `db2.cip.ifi.lmu.de`. Der gesamte String sollte also so aussehen: `DBI:mysql:database=$database;host=$host`. (Falls der Datenbankserver einen speziellen Port benutzt, sollte der auch noch angegeben werden, aber das ist bei uns nicht der Fall.)
2. Den Usernamen (in unserem Fall identisch mit dem Rechner-Login).
3. Das Passwort, das ihr bei der Einrichtung der Datenbank definiert habt.
4. *optional:* eine Hashreferenz, mit der verschiedene Optionen gesetzt werden können. Dabei besonders interessant:
   - `PrintError => 1` oder `RaiseError => 1`: damit wird eingestellt, dass bei Fehlern beim Datenbankzugriff die Fehlermeldungen nur ausgegeben werden (PrintError) oder gleich das Programm abbricht (RaiseError).
   - `AutoCommit => 0` oder `AutoCommit => 1`: Damit wird gesteuert, ob alle Zugriffe, die die Datenbank verändern, unmittelbar ausgeführt werden sollen (0) oder erst bei explizitem Aufruf eines `commit`-Befehls (1).

Der Rückgabewert dieses `DBI-connect()`-Aufrufs ist ein Objekt vom Typ `DBI`, auch Datenbank-Handle genannt, oder, bei Scheitern, ein `undef`. Fehlermeldungen sind grundsätzlich (auch bei Aufruf von anderen Methoden des DBI-Moduls) in der Variablen `$DBI::errstr` abgelegt – die man aber bei Setzen von `RaiseError` oder `PrintError` normalerweise nicht braucht.

Alles in allem könnte unser Programmanfang also so aussehen:

```
1 #!/usr/bin/perl
2
3 use strict;
4 use warnings;
```

```
5
6  use DBI;
7
8  my $database = "boskh_test";
9  my $host = "db2.cip.ifi.lmu.de";
10 my $username = "username"; # Usernamen einsetzen
11 my $database_password = "passwort"; # Passwort fuer die Datenbank
       einsetzen
12
13 my $dbh = DBI->connect("DBI:mysql:database=$database;host=$host",
       $username, $database_password, { RaiseError => 1, AutoCommit =>
       1 });
```

... und das Programmende:

```
1  $dbh->disconnect();
```

um die Verbindung wieder zu trennen. Fehlt nur noch ein bisschen was dazwischen.

Eintragen von Werten in die Datenbank

Es soll hier nicht näher darauf eingegangen werden, wie man eine Datenbank entwirft, sondern nur zu Demozwecken eine Tabelle mit Teilnehmern eines Kurses entworfen werden (während sich eine Datenbank normalerweise aus einer ganzen Reihe von Tabellen zusammensetzt). Auch sollen nicht die einzelnen SQL-Kommandos im Vordergrund stehen, sondern ihre Anwendung aus einem PERL-Programm heraus.

Im Shell-Client wäre der Befehl zum Anlegen einer Tabelle mit Kursteilnehmern z.B.

```
create table kursteilnehmer (name varchar(20), vorname varchar(20), matrikelnr char(12), gebdatum date);
```

`varchar` bedeutet dabei, dass die Einträge nicht alle dieselbe Länge haben müssen. Die Zahl dahinter in Klammern steht für die maximale Länge der Einträge – das kann man ggfs. auch nachträglich nochmal ändern. Die Matrikelnummer hingegen kann man als Typ `char` anlegen, wenn man annimmt, dass sie immer 12 Stellen lang ist. `date` ist ein eigener Datentyp in der Form `JJJJ-MM-TT`.

Es gibt zwei Wege, dasselbe von einem PERL-Programm aus zu erreichen. Zunächst die ausführlichere Methode:

1. Vorbereiten der SQL-Anweisung:

   ```
   1  my $sth = $dbh->prepare('create table kursteilnehmer (name
          varchar(20), vorname varchar(20), matrikelnr char(12),
          gebdatum date)');
   ```

   Rückgabewert ist ein sog. Statement Handler. Dieses Objekt beinhaltet jetzt die vorbereitete SQL-Anweisung.

2. Ausführen der SQL-Anweisung:

   ```
   1  $sth->execute();
   ```

3. Deaktivieren des Statement Handlers:

   ```
   1  $sth->finish();
   ```

   Um diesen Befehl jetzt von unserem PERL-Programm aus umzusetzen, können wir aber auch einfach die `do`-Methode benutzen, die die 3 obigen Befehle zusammenfasst:

   ```
   1  $dbh->do("create table kursteilnehmer (name varchar(20),
          vorname varchar(20), matrikelnr char(12), gebdatum date)");
   ```

Damit haben wir dann unsere Tabelle angelegt – jetzt sollte noch irgendwas darin eingetragen werden, z.B.

```
1 $dbh->do('insert into kursteilnehmer values ("Maier", "Sabine",
      "130780995543", "1980-07-13")');
2 $dbh->do('insert into kursteilnehmer values ("Schmitt", "Bernd",
      "190379634523", "1979-03-19")');
3 $dbh->do('insert into kursteilnehmer values ("Schuster","Michael
      ","010982940302", "1982-09-01")');
```

Grundsätzlich beachtet werden sollten die Anführungszeichen um die Feldwerte – und auch, dass der an `do` übergebene Wert ein einziger String sein sollte.

Warum benutzt man nicht immer die `do`-Methode? Das hat 2 Gründe:

1. Wenn man dieselben SQL-Anweisungen wiederholt ausführen möchte (z.B. in einer Schleife), dann ist es effizienter, die SQL-Anweisung einmal vorzubereiten und dann jeweils nur noch die vorbereitete Anweisung ausführen zu lassen.

2. Die `do`-Methode liefert grundsätzlich nur "true" oder "false" zurück. Für die Abfrage von Daten aus einer Tabelle ist sie daher ungeeignet.

Es hängt somit vom Kontext ab, wann das Benutzen der `do`-Methode sinnvoll ist und wann nicht. Werden z.B. die Daten, die in die Tabelle eingetragen werden sollen, aus einer Datei eingelesen, kann man den entsprechenden Befehl mit Platzhaltern (in Form von Fragezeichen) vorbereiten und dann jeweils mit den konkreten Daten ausführen lassen:

```
1  my $sth = $dbh->prepare('insert into kursteilnehmer values
       (?,?,?,?)');
2
3  open IN, "daten.txt" or die "$!";
4  while (<IN>) {
5    next if /^\s*$/;
6    chomp;
7    my ($name, $vorname, $matrikel, $gebdatum) = split(/\s+/,$_);
8    $sth->execute($name, $vorname, $matrikel, $gebdatum);
9  }
10 close IN or die "$!";
11
12 $sth->finish();
```

Dies ist um einiges effizienter als das wiederholte Aufrufen von `do`.

Abfragen der Datenbank mit einem Statement Handler

Wie bereits erwähnt, kann die `do`-Methode nicht zur Abfrage von Daten benutzt werden. Stattdessen kann man einen Statement Handler benutzen:

```
1 my $sth = $dbh->prepare('select * from kursteilnehmer');
2 $sth->execute();
3 while(my @row = $sth->fetchrow_array) {
4   print join(" ", @row), "\n";
5 }
6 $sth->finish();
```

Hier wird die Methode `fetchrow_array` benutzt, die jedesmal die nächste von `select` zurückgegebene Zeile als Array zurückgibt. Alternativ kann man die Methode `fetchrow_hashref` benutzen. Dabei wird die Zeile in ein Hash eingelesen, wobei die Keys jeweils die Spaltennamen sind und die Values die Werte der entsprechenden Zeile. Die Funktion gibt eine Referenz auf dieses Hash zurück:

```
1 my $sth = $dbh->prepare('select * from kursteilnehmer');
2 $sth->execute();
3 while(my $row_href = $sth->fetchrow_hashref) {
4   while (my ($key, $value) = each%{$row_href}) {
5     print "$key\t$value\n";
6   }
7   print "\n";
8 }
9 $sth->finish();
```

Abfragen der Datenbank ohne Statement Handler

Für die Abfrage von Einträgen gibt es wieder mehrere Methoden, die die Einzelschritte zusammenfassen:

- `selectrow_array`
- `selectrow_arrayref`
- `selectrow_hashref`
- `selectall_arrayref`
- `selectall_hashref`
- `selectcol_arrayref`

Dabei verleitet der Name `selectrow_array` vielleicht zu der Vermutung, man würde alle zutreffenden Zeilen einer Abfrage in einem Array zurückbekommen. Was man stattdessen bekommt, ist – in Listenkontext – die erste zutreffende Zeile als eine Liste mit den einzelnen Feldern als Elementen. In skalarem Kontext erhält man sogar nur den Inhalt des ersten Feldes der ersten zutreffenden Zeile. Ähnlich verhalten sich die anderen mit `selectrow_`-beginnenden Methoden.

Wenn man alle zutreffenden Zeilen auf einmal erhalten will, empfiehlt sich deshalb die Nutzung von `selectall_arrayref` bzw. `selectall_hashref`. Bei `selectall_arrayref` erhält man eine Referenz auf ein Array, das wiederum Referenzen auf Arrays enthält, in denen dann die Felder der einzelnen zutreffenden Zeilen stehen. Eine Abfrage aller Kursteilnehmer mit Geburtsdatum vor 1981 würde man also so formulieren:

```
1 my $all_hits_aref = $dbh->selectall_arrayref('select * from
    kursteilnehmer where gebdatum < "1981-01-01"');
2 foreach my $row_aref (@$all_hits_aref) {
3   print join("\t",@$row_aref), "\n";
4 }
```

Will man ohne weitere Einschränkung alle Kursteilnehmer anzeigen lassen, kann man bei obiger Anfrage den Teil ab `where` einfach weglassen.

Eine Alternative zu der Abfrage mit `selectall_arrayref` kann `selectall_hashref` sein. Hier wird eine Referenz auf ein Hash zurückgegeben, dessen Werte wiederum Hashes sind – eins pro Zeile, jeweils mit dem Spaltennamen als Schlüssel und dem Inhalt als Wert. Was der Schlüssel des übergeordneten Hashes sein soll, muss man beim Aufruf mitgeben.

Nehmen wir an, wir wollten dieselbe Anfrage durchführen wie mit `selectall_arrayref`, dann aber kontrollieren, ob sich unter den vor 1981 geborenen jemand mit dem Namen "Schmitt" befindet. Das könnte mit `selectall_hashref` dann so aussehen:

```
1 my $all_hits_href = $dbh->selectall_hashref('select * from
    kursteilnehmer where gebdatum < "1981-01-01"', 'name');
2 if ( my $schmitt_href = ${$all_hits_href}{Schmitt} ) {
3   print "Unter den vor 1981 geborenen gibt es tatsaechlich jemanden
        namens Schmitt.\n";
4   print "Ihr / Sein Geburstdatum ist: ", ${$schmitt_href}{gebdatum
      }, "\n";
5 } else {
6   print "Unter den vor 1981 geborenen gibt es niemanden namens
      Schmitt.\n";
7 }
```

Wie man sehen kann, ist bei dieser Methode der Vorteil, dass man nach über die Hash-Schlüssel ganz gezielt auf gewünschte Datensätze zurückgreifen kann.

Als letzte der o.g. Methoden verbleibt noch `selectcol_arrayref`. Diese Methode liefert eine Referenz auf ein Array zurück, dass alle Werte einer Spalte enthält. Dabei wird per default von jeder Rückgabezeile nur der Eintrag in der ersten Spalte gewählt. Entscheidend ist dabei nicht, welche Spalten es in der abgefragten Tabelle gibt, sondern welche Spalten vom SQL-Befehl zurückgegeben werden. Da man mit der `select`-Anweisung beliebige Spalten aus einer Tabelle auswählen kann, kann man darüber steuern, welche Spalte aus der Datenbank-Tabelle die erste ist in der von select zurückgegebenen Datenstruktur – und deren Werte finden sich dann im Array, das von `selectof_arrayref` zurückgegeben wird. Insgesamt *filtert* man also zweimal hintereinander: einmal mit `select` aus der Datenbank-Tabelle, und aus dem, was da als Ergebnis zurückkommt, filtert man nochmal etwas mit `selectcol_arrayref` heraus.

Alle Matrikelnummern der Kursteilnehmer würde man sich also wie folgt anzeigen lassen:

```
1 my $first_col_aref = $dbh->selectcol_arrayref('select matrikelnr
    from kursteilnehmer');
2 foreach my $matrikelnr (@$first_col_aref) {
3   print "$matrikelnr\n";
4 }
```

Das Ergebnis bleibt dasselbe, wenn man sich mit select mehrere Spalten zurückgeben lässt – in `$first_col_aref` landen auch in diesem Fall nur die Werte der ersten von `select` zurückgegebenen Spalten:

```
1 my $first_col_aref = $dbh->selectcol_arrayref('select matrikelnr,
    name, vorname from kursteilnehmer');
2 foreach my $matrikelnr (@$first_col_aref) {
3   print "$matrikelnr\n";
4 }
```

Mit dem zusätzlichen Attribut `Columns` kann man aber auch erreichen, dass die Inhalte mehrerer Spalten gleichzeitig zurückgegeben werden. Attribute werden bei den Abfragebefehlen zusammen mit ihren Belegungen innerhalb einer Hashreferenz als zweites Argument übergeben, also wie das vierte Argument beim `connect`-Befehl. Der Wert zu `Columns` muss eine Arrayreferenz sein, und die Inhalte des Arrays müssen die Nummern derjenigen Spalten sein, deren Werte innerhalb des `select`-Rückgabewertes in das von `selectcol_arrayref` zurückgegebene Array aufgenommen werden sollen. Die Zählung beginnt dabei bei 1, nicht etwa bei 0. Wollte man sich z.B. von der obigen Anfrage sowohl die Matrikelnummer als auch den Vornamen in das zurückgegebene Array übertragen lassen, müsste man das so schreiben:

```
1 my $two_col_aref = $dbh->selectcol_arrayref('select matrikelnr,
    name, vorname from kursteilnehmer', {Columns=>[1,3]});
2 foreach my $value (@$two_col_aref) {
3   print "$value\n";
```

```
4 }
```

Die Ausgabe dieses Programmabschnitts wären abwechselnd je eine Zeile mit einer Matrikelnummer und dann eine Zeile mit dem zugehörigen Vornamen.

So ein Rückgabearray mit Werten aus 2 Spalten lässt sich wunderbar leicht in ein Hash umwandeln. Um z.B. ein Hash zu erzeugen, dass als Schlüssel alle Matrikelnummern enthält und als Werte die zugehörigen Namen, könnte man folgendes schreiben:

```
1 my $two_cols_aref = $dbh->selectcol_arrayref('select matrikelnr,
     name from kursteilnehmer', {Columns=>[1,2]});
2 my %matrikelnummern = @$two_cols_aref;
3
4 while (my ($matrikel, $name) = each%matrikelnummern) {
5   print "$matrikel\t$name\n";
6 }
```

Löschen von Datenbankeinträgen

Falls man mal Datensätze aus einer Datenbanktabelle löschen will, kann man das mit dem SQL-Befehl `delete` machen. Da keine Daten zurückgegeben werden, kann man diesen Befehl in eine `do`-Anweisung verpacken:

```
1 $dbh->do('delete from kursteilnehmer');
```

Damit würde man dann den kompletten Inhalt der Datenbanktabelle löschen (wobei die Tabelle als solche aber erhalten bleibt, nur Inhalt hat sie dann keinen mehr). Um nur ausgewählte Datensätze zu löschen, kann man mittels `where` wieder Einschränkungen definieren.

Will man die Tabelle als solches löschen, lautet der Befehl

```
1 $dbh->do('drop table kursteilnehmer');
```

22.7 Übungsaufgaben

Aufgabe 22.1 Ändere die Lösung zu Aufgabe 21.3 so ab, dass die Bücher-Daten aus einer DBM-Datei eingelesen werden.

Hinweise:

- Für ernsthafte Anwendungen sollte in so einem Fall mit File Locking gearbeitet werden, da ja gerade bei CGI-Programmen zeitgleiche Zugriffe verschiedener Prozesse auf die Daten vorkommen können. Für die vorliegende Übungsaufgabe kann jedoch darauf verzichtet werden.
- Da das Modul `MLDBM` keine Arrays, sondern nur Hashes verarbeitet, muss die Datenstruktur entsprechend angepasst werden.

Aufgabe 22.2 Richte eine Datenbank ein und schreibe ein PERL-Programm, das eine Tabelle anlegt, Werte darin einträgt und diese dann abfragt (mit oder ohne Einschränkung der Auswahl) und anzeigt.

23

PHP als Alternative zu PERL

23.1 Überblick

Datenbankabfragen und die Bearbeitung von Webformularen kann man natürlich innerhalb eines PERL-CGI-Programms machen, und das ist auch eine sehr häufige Kombination – bzw. ist es bis jetzt gewesen. Mittlerweile werden diese Aufgaben immer öfter mit PHP gelöst. Deshalb soll hier auch kurz auf PHP als Alternative zu PERL eingegangen werden.

Einer der Vorteile von PHP ist, dass man Programmcode direkt in HTML-Code einbinden kann, statt den HTML-Code erst per CGI-Skript erzeugen zu müssen.

Man kann also z.B. eine Webseite schreiben mit

```
<html>
<head>
<title>PHP-Testseite</title>
</head>
<body>
<script language="php">
echo "hallo";
</script>
</body>
</html>
```

Das eingefügte PHP-Kommando `echo` funktioniert analog zu `print` in PERL und sorgt dafür, dass bei Aufruf der Webseite der Text "hallo" erscheint. Wird in diesem Rahmen HTML-Code ausgegeben (z.B. mit `echo "<h1>hallo</h1>";`), dann werden diese HTML-Anweisungen vom Browser berücksichtigt. Solche PHP-Seiten werden mit der Endung ".php" statt ".html" versehen und im Verzeichnis $HOME/public_html/php gespeichert.

Das Ganze hätte man auch in Kurzform so schreiben können:

```
<html>
<head>
<title>PHP-Testseite</title>
</head>
<body>
<?php echo "hallo"; ?>
</body>
</html>
```

Dabei kann innerhalb eines einzelnen Dokuments beliebig zwischen beiden Möglichkeiten (und außerdem auch reinem HTML-Code) gewechselt werden; nur darf ein PHP-Skript nicht im head-Teil anfangen und dann erst im body-Teil enden. Die in verschiedenen Teilen des Webdokuments geschriebenen PHP-Anweisungen bilden ein und dasselbe Programm, auch wenn sie durch HTML-Code o.ä. unterbrochen werden.

Kommentare

Zum Einfügen von einzeiligen Kommentaren dient ein doppelter Slash (//), alles zwischen einem solchen doppelten Slash und dem Zeilenende wird als Kommentar gewertet. Mehrzeilige Kommentare müssen zwischen /∗ und ∗/ eingeschlossen werden. Das in Perl verwendete # findet als Kommentarzeichen keine Anwendung.

Variablen

Bei den Variablen gibt es einige Gemeinsamkeiten mit PERL: sie brauchen nicht vorab deklariert zu werden, und wie skalare Variablen in PERL beginnen sie mit einem Dollarzeichen (in PHP beginnen allerdings *alle* Variablen mit einem Dollarzeichen, nicht nur skalare Variablen).

Hier ein Beispiel für ein PHP-Programm mit Variablen und Kommentaren:

```
<html>
<head>
<title>PHP-Testseite</title>
</head>
<body>
<script language="php">
$a = 7; // Kommentar
$b = 13.3;
$c = $a + $b;
echo $c;
/* längerer Kommentar
über mehrere
Zeilen hinweg */
</script>
</body>
</html>
```

Intern kennt PHP folgende Datentypen:

- Strings
- ganze Zahlen
- Zahlen mit Nachkommastellen
- Arrays (dazu werden auch assoziative Arrays, sprich Hashes, gezählt)
- Objekte

Meistens geschehen nötige Umwandlungen des Datentyps (z.B. von einer ganzen Zahl in eine Zahl mit Nachkommastellen) zum Glück aus dem Kontext heraus.

Arrays werden wie folgt angelegt:

```
1 $a = array(7,3,5,1);
```

bzw.

```
1 $a[0] = 7;
2 $a[1] = 3;
3 ...
```

und Hashes:

```
1 $h = array("a" => 7, "b" => 3, "c" => 5, "d" => 1);
```

bzw.

```
1  $h["a"] = 7;
2  $h["b"] = 3;
3  ....
```

Funktionen

Subroutinen werden in PHP "Funktionen" genannt und entsprechend mit `function` statt mit `sub` deklariert. Soll eine Funktion Parameter entgegennehmen, dann müssen diese schon bei der Deklaration der Funktion angegeben werden, z.B.

```
1  function hypothenuse ($a,$b) {
2    ....
3    ....
4    return $c;
5  }
```

Dabei kann man auch Referenzen (markiert durch ein vorangestelltes &) verwenden, dann werden von den im Aufruf verwendeten Variablen direkt Referenzen erzeugt.

Funktionen werden üblicherweise im head-Bereich einer HTML-Seite deklariert.

Gültigkeitsbereich von Variablen

Globale Variablen müssen außerhalb von allen Funktionen deklariert werden. Sie sind auch nur außerhalb von Funktionen gültig. Um sie dennoch innerhalb einer Funktion verwenden zu können, müssen sie dort entweder mit dem Schlüsselwort *global* eingeführt werden, oder sie müssen als Parameter an die Funktion übergeben werden.

Variablen, die innerhalb einer Funktion verwendet werden und nicht mit *global* definiert wurden, sind automatisch lokal, d.h. nur in dieser Funktion gültig. Dies gilt auch für die Variablen, die die übergebenen Parameter aufnehmen.

Das ist zusammengenommen also ungefähr so, als würden in PERL alle Variablen, die außerhalb von Subroutinen definiert werden, automatisch mit `our` versehen und alle, die innerhalb von Subroutinen definiert werden, mit `my`.

Beispiel:

```
1  <html>
2  <head>
3  <title>PHP-Testseite</title>
4  <script language="php">
5
6  function mult ($a,$b) {
7    echo $a * $b * $c, "<p>";
8    global $c;
9    echo $a * $b * $c, "<p>";
10 }
11
12 </script>
13 </head>
14 <body>
15 <script language="php">
16
17 $c = 7;
```

```
18 mult(3,2);
19
20 </script>
21 </body>
22 </html>
```

Die erste Ausgabe erzeugt nur eine Null, da $c an dieser Stelle noch keinen Wert hat. Erst mit `global $c` wird der Wert aus dem Hauptprogramm in der Funktion sichtbar.

Vergleichsoperatoren

Was die Vergleichsoperatoren betrifft, kann man in PHP auch Strings mit == und != vergleichen; man braucht also kein gesondertes `eq` oder `ne` o.ä. wie in Perl. Die Operatoren <, >, <=, >= hingegen sind nur auf Zahlen anwendbar.

Kontrollstrukturen

Bei den Kontrollstrukturen gibt es nur kleinere Abweichungen von PERL:

- statt `elsif` muss man `elseif` schreiben
- es gibt eine `switch`-Anweisung für eine Mehrfachauswahl
- statt `next` und `last` heißen die entsprechenden Kommandos `continue` und `break`
- alternativ zu den geschweiften Klammern kann man bei `if` (und entsprechend bei `switch`) folgende Schreibweise verwenden:

```
1 if ($a<7):
2   echo "kleiner!<p>";
3   // ggfs. weitere Anweisungen
4 endif;
```

Wenn nur eine Zeile im Inneren der Kontrollstruktur steht, kann man (wie in C) die geschweiften Klammern auch weglassen:

```
1 if ($a<7)
2   echo "kleiner!<p>";
```

Die Syntax einer `foreach`-Schleife ist leicht anders als in PERL:

```
1 $a = array(7,3,5,1); // ein normales Array
2
3 foreach ($a as $y) {
4   echo $y, "<p>\n";
5 }// gibt nacheinander die Werte des Arrays aus
6
7 foreach ($a as $x => $y) {
8   echo $x." ".$y, "<p>\n";
9 }// gibt nacheinander jeweils Index und zugehoerigen Wert des
     Arrays aus
```

Bei einem Hash funktioniert das ähnlich, nur werden in der zweiten Variante statt der Indizes die Schlüssel abgefragt:

```
1 $a = array("a" => 7, "b" => 3, "c" => 5, "d" => 1);  // ein Hash
2
3 foreach ($a as $y) {
4   echo $y, "<p>\n";
5 }// gibt nacheinander die Werte des Hashes aus
6
```

```
7  foreach ($a as $x => $y) {
8    echo $x." ".$y, "<p>\n";
9  }// gibt nacheinander jeweils Key und zugehoerigen Wert des Hashes
       aus
```

Wer sich näher mit PHP beschäftigen möchte, findet ein gutes Online-Tutorial unter `http://www.galileocomputing.de/openbook/php4/index.htm`.

23.2 Bearbeitung von Webformularen mit PHP

Hat man ein Webformular geschrieben, in dem jemand seinen Namen und einen Kommentar hinterlassen kann, und will man diese beiden Felder mit einem CGI-Skript nochmal darstellen, dann würde das Skript ungefähr so aussehen:

```
1  #!/usr/bin/perl
2
3  use strict;
4  use warnings;
5  use CGI qw(:standard escapeHTML);
6
7  BEGIN {
8    use CGI::Carp qw(fatalsToBrowser carpout);
9    open (LOG, ">>errors.log") or die "couldn't open log file: $!";
10   carpout(*LOG);
11 }
12
13 my $user = param('AnwenderName');
14 my $comment = param('Kommentartext');
15
16 print header();
17 print start_html("Kommentar-Echo");
18 print p(escapeHTML($user)." schrieb:");
19 print p(escapeHTML($comment));
20 print end_html();
```

In PHP braucht man keine `param`-Methode: die Inhalte stehen sofort in der Hashtabelle `$_REQUEST[name]` zur Verfügung; die Methode `escapeHTML` heißt bei PHP `htmlentities`:

```
<html>
<head>
<title>Kommentar-Echo</title>
</head>
<body>
<script language="php">

echo htmlentities($_REQUEST['AnwenderName']), " schrieb:<p>";
echo htmlentities($_REQUEST['Kommentartext']), "<p>"

</script>
</body>
</html>
```

23.3 Datenbankabfrage mit PHP

Für die Abfrage von Datenbanken gibt es eine ganze Reihe bereits implementierter Befehle. Hier ein grundlegender Programmablauf:

Verbindungsaufbau

Der Verbindungsaufbau zu einer MySQL-Datenbank erfolgt so:

```
1 $dbh = mysql_connect("hostname", "kennung", "passwort");
```

Einfügen von Datensätzen

Nach dem Verbinden können Datensätze in eine Tabelle eingefügt werden, und zwar mit dem Befehl `mysql_db_query` (in $db soll dabei der Name der Datenbank stehen):

```
1 mysql_db_query($db, 'insert into kursteilnehmer values ("Schuster",
      "Michael","010982940302","1982-09-01")');
```

Wenn man überprüfen will, ob das Einfügen erfolgreich war, ruft man die Funktion `mysql_affected_rows` auf. Sie liefert die Anzahl der betroffenen Datensätze, wenn Befehle ausgeführt wurden, die Datensätze verändern (z.B. einfügen, ändern, löschen, aber nicht bei reinen Abfragen). Liefert sie nach unserer Einfügeoperation einen Wert größer Null, ist alles ok, andernfalls gab es ein Problem:

```
1 $affected = mysql_affected_rows();
2
3 if ($affected)
4   echo "Datensatz erfolgreich eingefuegt!<br>\n";
5 else
6   echo "Irgendwas ist schiefgelaufen beim Einfuegen des Datensatzes
      !<br>\n";
```

Abfragen von Datensätzen

Das Abfragen von Datensätzen erfolgt mit demselben Befehl wie das Einfügen: mit `mysql_db_query`. Allerdings fragt man hier direkt den Rückgabewert ab, in dem das Ergebnis gespeichert ist:

```
1 $result = mysql_db_query($db, "select * from kursteilnehmer");
```

Im Falle einer reinen Abfrage, bei der keine Datensätze verändert werden, kann man die Anzahl der gefundenen Datensätze mit der Funktion `mysql_num_rows` herausfinden, der das Ergebnis übergeben werden muss:

```
1 $num_rows = mysql_num_rows($result);
2 echo "Es wurden $num_rows Datensaetze gefunden!<br>\n";
```

Ausgabe des Abfrageergebnisses

Die gefundenen Datensätze kann man sich, wie auch in PERL mit dem DBI-Modul, auf mehrere verschiedenen Weisen zurückgeben lassen – bei PHP aber normalerweise immer aus dem Rückgabewert von `mysql_db_query`.

Eine der Möglichkeiten ist `mysql_result`. Damit kann man sich ganz gezielt einzelne Feldwerte aus dem Ergebnis geben lassen. Die 3 Argumente sind: die von `mysql_db_query` zurückgegebene Ergebnisvariable, Index des Datensatzes innerhalb des Ergebnisses (beginnend bei 0), Feldname. Man könnte sich also das Ergebnis der obigen Abfrage wie folgt anzeigen lassen:

```
1 for ($i=0; $i<$num_rows; $i++) {
2
3   echo mysql_result($result, $i, "name"), ", ";
4   echo mysql_result($result, $i, "vorname"), ", ";
5   echo mysql_result($result, $i, "matrikelnr"), ", ";
6   echo mysql_result($result, $i, "gebdatum"), "<br>\n";
7
8 }
```

Pro Schleifendurchlauf wird dabei ein Datensatz ausgegeben, wobei die einzelnen Anweisungen jeweils genau ein Feld des Datensatzes abfragen.

Ähnlich wie bei PERL/DBI gibt es aber wieder einige komfortablere Befehle, wenn man nicht jedes Datenfeld einzeln abfragen will. So liefert z.B. `mysql_fetch_row` jeweils den nächsten Datensatz aus der Ergebnisvariablen als Array zurück:

```
1  $num_cols = 4;
2
3  for ($i = 0; $i < $num_rows; $i++) {
4
5    $array = mysql_fetch_row($result);
6
7    for ($j = 0; $j < $num_cols; $j++)
8      echo $array[$j], " ";
9
10   echo "<p>";
11 }
```

Eine andere Möglichkeit ist es, sich die Ergebnisse mit `mysql_fetch_assoc` als Hash zurückgeben zu lassen. Damit sähe dann die Ausgabe der Datensätze z.B. so aus:

```
1  while($row_as_hash = mysql_fetch_assoc($result)) {
2    foreach ($row_as_hash as $x => $y) {
3      echo "$x: $y<br>\n";
4    }
5    echo "<p>\n";
6  }
```

Schließen der Datenbank-Verbindung

Zum Schließen der Datenbank-Verbindung braucht man wieder den Rückgabewert der `connect`-Anweisung:

```
1  mysql_close($dbh);
```

Anmerkung: in PHP5 wurde ein neues Datenbankmodul mysqli implementiert, welches ein prozedurales Interface besitzt, das fast mit dem des vorgestellten Moduls mysql identisch ist. Zusätzlich besitzt mysqli ein objektorientiertes Interface und einige Funktionserweiterungen.